ΘΕΩΡΙΑ ΚΑΙ ΑΣΚΗΣΕΙΣ ΣΤΗΝ POSTGRESQL 18

SQL

με PostgreSQL

Από το πρώτο SELECT μέχρι το EXPLAIN ANALYZE.

41 **κεφάλαια, 253 ασκήσεις**

ΚΑΘΕ QUERY ΕΚΤΕΛΕΣΤΗΚΕ ΣΤΗΝ POSTGRESQL 18.6

ΠΡΙΝ ΑΠΟ ΤΟ ΠΡΩΤΟ QUERY

Η SQL περιγράφει το αποτέλεσμα. Η βάση διαλέγει τον δρόμο.

Γράφεις τι θέλεις να πάρεις πίσω και η PostgreSQL αποφασίζει πώς θα το βρει. Αυτή η ιδέα εξηγεί γιατί η SQL μοιάζει απλή στα πρώτα queries και γιατί γίνεται απαιτητική αργότερα. Για να γράφεις σωστά queries αρκεί να καταλαβαίνεις τι σημαίνουν. Για να γράφεις γρήγορα queries πρέπει να καταλαβαίνεις και πώς εκτελούνται.

Το βιβλίο προχωρά με αυτή τη σειρά. Ξεκινά από έναν πίνακα και ένα SELECT. Περνά σε joins, aggregates και subqueries, μετά σε αλλαγές δεδομένων, σχήμα και transactions. Στο τέταρτο μέρος φτάνει σε CTE, recursive queries, window functions, JSONB και αναζήτηση κειμένου με trigrams. Τα δύο τελευταία μέρη ανοίγουν τη μηχανή. Indexes, query plans, EXPLAIN ANALYZE, statistics, MVCC, locks και partitioning.

Θεωρεί ότι ξέρεις να δουλεύεις σε terminal και ότι έχεις γράψει λίγο κώδικα σε κάποια γλώσσα. Δεν θεωρεί ότι έχεις γράψει SQL. Κάθε έννοια εμφανίζεται πρώτα με μικρό παράδειγμα και χρησιμοποιείται ξανά αργότερα, ώστε να μη χρειάζεται να τη θυμάσαι από άλλη πηγή.

Πώς δουλεύεται

Όλα τα κεφάλαια χρησιμοποιούν την ίδια βάση, ένα μικρό ηλεκτρονικό κατάστημα με πελάτες, προϊόντα, παραγγελίες και πληρωμές. Το κεφάλαιο 1 δείχνει πώς τη στήνεις. Κάθε query του βιβλίου εκτελέστηκε στην PostgreSQL 18.6 και το αποτέλεσμα κάτω από αυτό είναι το πραγματικό. Αν το τρέξεις στη δική σου βάση πρέπει να δεις τις ίδιες γραμμές. Εξαίρεση είναι οι χρόνοι και τα κόστη στα query plans, που εξαρτώνται από το μηχάνημα.

Κάθε κεφάλαιο κλείνει με ασκήσεις που χρησιμοποιούν μόνο όσα έχουν ήδη παρουσιαστεί. Κάτω από κάθε εκφώνηση υπάρχει το αναμενόμενο αποτέλεσμα, ώστε να ελέγχεις το query σου χωρίς να κοιτάξεις τη λύση. Οι λύσεις με εξηγήσεις βρίσκονται στον χωριστό τόμο.

Δεν απομνημονεύουμε συνταγές. Κάθε query διαβάζεται ως περιγραφή ενός συνόλου γραμμών και κάθε plan ως απόφαση που μπορείς να εξηγήσεις.

Ο ΧΑΡΤΗΣ ΤΟΥ ΒΙΒΛΙΟΥ

Περιεχόμενα

ΜΕΡΟΣ Ι

Ένας πίνακας

Πριν ενώσουμε πίνακες μαθαίνουμε να ρωτάμε σωστά έναν. Ποιες γραμμές, ποιες στήλες, με ποια σειρά. Τι σημαίνει μια τιμή που λείπει και πώς δουλεύουν οι αριθμοί, το κείμενο και ο χρόνος.

ΚΕΦΑΛΑΙΟ 1

Βάση, πίνακες και το πρώτο SELECT

Μια βάση δεδομένων κρατά πίνακες. Ένα query περιγράφει ποιες γραμμές και ποιες στήλες θέλεις να πάρεις πίσω. Σε αυτό το κεφάλαιο στήνεις τη βάση του βιβλίου και γράφεις τα πρώτα σου SELECT.

Πίνακες, γραμμές και στήλες

Η PostgreSQL είναι σχεσιακή βάση δεδομένων. Οργανώνει τα δεδομένα σε **πίνακες** (tables). Κάθε πίνακας έχει σταθερό σύνολο από **στήλες** (columns) και κάθε στήλη έχει όνομα και **τύπο**. Η στήλη `price` είναι αριθμός με δύο δεκαδικά, η `name` είναι κείμενο, η `created_at` είναι χρονική στιγμή. Οι **γραμμές** (rows) είναι οι εγγραφές. Μια γραμμή του πίνακα `customers` είναι ένας πελάτης.

Δύο ιδιότητες των πινάκων θα μας απασχολούν σε όλο το βιβλίο. Η πρώτη είναι ότι οι γραμμές δεν έχουν σειρά. Ένας πίνακας είναι σύνολο γραμμών, όχι λίστα. Αν θέλεις σειρά, τη ζητάς ρητά. Η δεύτερη είναι ότι κάθε γραμμή πρέπει να ξεχωρίζει από τις υπόλοιπες μέσω ενός **primary key**. Στους περισσότερους πίνακες του βιβλίου αυτό είναι η στήλη `id`.

Η SQL είναι η γλώσσα με την οποία μιλάς στη βάση. Είναι **δηλωτική**. Δεν γράφεις βήματα όπως σε Python ή Ruby. Περιγράφεις το αποτέλεσμα που θέλεις και η PostgreSQL αποφασίζει πώς θα το υπολογίσει. Αυτή η απόφαση λέγεται **query plan** και θα την ανοίξουμε στο πέμπτο μέρος του βιβλίου.

Στήσιμο της βάσης

Χρειάζεσαι PostgreSQL 18. Σε macOS τη βάζεις με Homebrew, σε Linux από το επίσημο repository της PostgreSQL για τη διανομή σου. Μαζί έρχεται το `psql`, ο client γραμμής εντολών που χρησιμοποιούμε σε όλο το βιβλίο.

Τα αρχεία του βιβλίου βρίσκονται στον φάκελο `data`. Το `schema.sql` φτιάχνει τους πίνακες, το `seed.sql` βάζει τα δεδομένα και το `lab.sql` φτιάχνει τους μεγάλους πίνακες που χρειάζονται από το κεφάλαιο 29 και μετά. Το `setup.sh` τα φορτώνει όλα με τη σωστή σειρά σε μια καθαρή βάση με όνομα `sqlbook`.

TERMINAL

```
bash data/setup.sh
psql -d sqlbook
```

Το script φτιάχνει τη βάση με ελληνικό collation, ώστε η ταξινόμηση κειμένου να ακολουθεί το ελληνικό αλφάβητο. Ορίζει επίσης ζώνη ώρας `Europe/Athens`. Και τα δύο επηρεάζουν αποτελέσματα που θα δεις στο βιβλίο. Όποτε θέλεις να ξεκινήσεις από την αρχή, για παράδειγμα αφού αλλάξεις δεδομένα στο κεφάλαιο 15, τρέχεις ξανά το ίδιο script.

Το psql

Το `psql` δέχεται δύο είδη εντολών. Οι εντολές SQL τελειώνουν με ερωτηματικό και στέλνονται στον server. Οι **meta εντολές** ξεκινούν με backslash και τις εκτελεί το ίδιο το `psql`. Η `\dt` δείχνει τους πίνακες της βάσης.

PSQL

```
\dt
```

ΑΠΟΤΕΛΕΣΜΑ

```
                List of tables
 Schema |       Name         | Type  |  Owner
--------+--------------------+-------+----------
 public | categories         | table | postgres
 public | cities             | table | postgres
 public | customers          | table | postgres
 public | employees          | table | postgres
 public | events             | table | postgres
 public | order_items        | table | postgres
 public | orders             | table | postgres
 public | payments           | table | postgres
 public | product_suppliers  | table | postgres
 public | products           | table | postgres
 public | reviews            | table | postgres
 public | suppliers          | table | postgres
(12 rows)
```

Η `\d` ακολουθούμενη από όνομα πίνακα δείχνει τις στήλες, τους τύπους και τους κανόνες του πίνακα.

PSQL

```
\d cities
```

ΑΠΟΤΕΛΕΣΜΑ

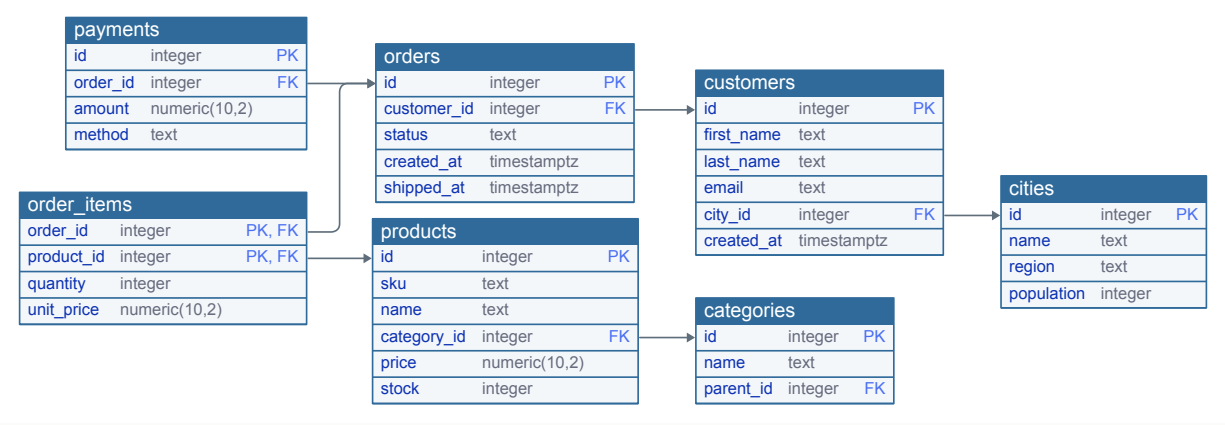
```
                Table "public.cities"
   Column   |  Type   | Collation | Nullable | Default
------------+---------+-----------+----------+---------
 id         | integer |           | not null |
 name       | text    |           | not null |
 region     | text    |           | not null |
 population | integer |           |          |
Indexes:
    "cities_pkey" PRIMARY KEY, btree (id)
    "cities_name_key" UNIQUE CONSTRAINT, btree (name)
Check constraints:
    "cities_population_check" CHECK (population > 0)
Referenced by:
    TABLE "customers" CONSTRAINT "customers_city_id_fkey" FOREIGN KEY (city_id) REFERENCES cities(id)
    TABLE "orders" CONSTRAINT "orders_shipping_city_id_fkey" FOREIGN KEY (shipping_city_id)
REFERENCES cities(id)
    TABLE "suppliers" CONSTRAINT "suppliers_city_id_fkey" FOREIGN KEY (city_id) REFERENCES cities(id)
```

Διάβασε το αποτέλεσμα από πάνω προς τα κάτω. Ο πίνακας έχει τέσσερις στήλες. Οι τρεις είναι `not null`, δηλαδή πρέπει πάντα να έχουν τιμή. Το `id` είναι primary key και το `name` είναι unique. Στο τέλος φαίνεται ότι τρεις άλλοι πίνακες δείχνουν στο `cities` μέσω foreign key. Αυτές οι σχέσεις είναι το θέμα του κεφαλαίου 7.

Τρεις ακόμη meta εντολές αξίζει να θυμάσαι από τώρα. Η `\x` εναλλάσσει την κάθετη εμφάνιση, χρήσιμη όταν μια γραμμή έχει πολλές στήλες. Η `\timing` τυπώνει πόσο κράτησε κάθε query. Η `\pset null 'ø'` εμφανίζει το σύμβολο ∅ όπου λείπει τιμή. Το βιβλίο τυπώνει τις τιμές που λείπουν με αυτό το σύμβολο και το κεφάλαιο 3 εξηγεί γιατί η διάκριση έχει σημασία. Βάλε τη γραμμή `\pset null 'ø'` στο αρχείο `~/.psqlrc` για να ισχύει πάντα.

Το σχήμα του βιβλίου

Η βάση περιγράφει ένα μικρό ηλεκτρονικό κατάστημα. Οι πελάτες ζουν σε πόλεις και κάνουν παραγγελίες. Κάθε παραγγελία έχει γραμμές, μία για κάθε προϊόν που αγοράστηκε. Τα προϊόντα ανήκουν σε κατηγορίες και οι πληρωμές αφορούν παραγγελίες.



Σχήμα 1.1 · Οι βασικοί πίνακες της βάσης. Κάθε βέλος ξεκινά από foreign key και δείχνει στο primary key που αναφέρει.

Υπάρχουν και πίνακες που δεν χωρούν στο σχήμα. Ο `employees` κρατά τους υπαλλήλους του καταστήματος και τον προϊστάμενο του καθενός. Ο `suppliers` και ο `product_suppliers` λένε ποιος προμηθευτής φέρνει ποιο προϊόν. Ο `reviews` κρατά κριτικές πελατών και ο `events` καταγράφει τι έκανε κάθε πελάτης στο site. Θα τους χρησιμοποιήσουμε όταν χρειαστούν.

SELECT και FROM

Το πιο απλό query ζητά όλες τις στήλες ενός πίνακα. Το αστεράκι σημαίνει όλες τις στήλες, με τη σειρά που ορίστηκαν.

```sql
SELECT * FROM cities;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |    name     |       region       | population
----+-------------+--------------------+------------
  1 | Αθήνα       | Αττική             |     643452
  2 | Θεσσαλονίκη | Κεντρική Μακεδονία |     309617
  3 | Πάτρα       | Δυτική Ελλάδα      |     167446
  4 | Ηράκλειο    | Κρήτη              |     179302
  5 | Λάρισα      | Θεσσαλία           |     148562
  6 | Βόλος       | Θεσσαλία           |     144449
  7 | Ιωάννινα    | Ήπειρος            |     113094
  8 | Χανιά       | Κρήτη              |     108642
  9 | Καλαμάτα    | Πελοπόννησος       |      69849
 10 | Ρόδος       | Νότιο Αιγαίο       |      49541
 11 | Κοζάνη      | Δυτική Μακεδονία   |      71388
(11 rows)
```

Κάτω από τις γραμμές το `psql` γράφει πόσες είναι. Οι αριθμοί στοιχίζονται δεξιά και το κείμενο αριστερά. Αυτή η μορφή είναι του `psql`. Η βάση επιστρέφει γραμμές και στήλες, όχι γραμμένο κείμενο.

Συνήθως δεν θέλεις όλες τις στήλες. Τις γράφεις μετά το `SELECT`, χωρισμένες με κόμμα, με όποια σειρά θέλεις να εμφανιστούν.

SQL

```sql
SELECT name, price, stock FROM products;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                |  price  | stock
------------------------------------+---------+-------
 Βίος και πολιτεία του Αλέξη Ζορμπά |   16.90 |    42
 Το Κιβώτιο                         |   14.50 |    18
 Η Φόνισσα                          |    9.90 |    55
 Ματωμένα χώματα                    |   15.20 |     0
 Μαθαίνω Python                     |   32.00 |    25
 Ruby από την αρχή                  |   29.50 |     9
 Καθαρός κώδικας στην πράξη         |   38.00 |    14
 PostgreSQL σε βάθος                |   45.00 |    11
 Σχεδίαση βάσεων δεδομένων          |   36.00 |     7
 Δίκτυα υπολογιστών                 |   41.00 |     6
 Laptop Aero 14                     | 1149.00 |     8
 Laptop Pro 16                      | 2199.00 |     4
 Laptop Student 15                  |  649.00 |    15
 Ασύρματο ποντίκι                   |   24.90 |   120
 …
(30 rows)
```

Το αποτέλεσμα έχει 30 γραμμές και το βιβλίο δείχνει τις πρώτες. Όταν μια έξοδος κόβεται, το βιβλίο γράφει τρεις τελείες και μετά το πλήθος των γραμμών που επέστρεψε πραγματικά το query.

ΠΑΓΙΔΑ

Το `SELECT *` βολεύει για να δεις γρήγορα έναν πίνακα. Σε κώδικα εφαρμογής γράψε τις στήλες που χρειάζεσαι. Αν κάποιος προσθέσει αργότερα στήλη με μεγάλο κείμενο, το `SELECT *` θα τη μεταφέρει σε κάθε query χωρίς να το ζητήσεις.

Εκφράσεις και aliases

Στη λίστα του `SELECT` δεν γράφεις μόνο ονόματα στηλών. Γράφεις **εκφράσεις**. Μια έκφραση υπολογίζεται μία φορά για κάθε γραμμή και δίνει μια τιμή. Η αξία του αποθέματος ενός προϊόντος είναι η τιμή επί την ποσότητα.

SQL

```
SELECT name, price, stock, price * stock
FROM products
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                | price | stock | ?column?
------------------------------------+-------+-------+----------
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90 |    42 |   709.80
 Το Κιβώτιο                         | 14.50 |    18 |   261.00
 Η Φόνισσα                          |  9.90 |    55 |   544.50
 Ματωμένα χώματα                    | 15.20 |     0 |     0.00
 Μαθαίνω Python                     | 32.00 |    25 |   800.00
(5 rows)
```

Το `LIMIT 5` κρατά μόνο πέντε γραμμές. Το χρησιμοποιούμε εδώ για να μένουν μικρά τα αποτελέσματα. Το κεφάλαιο 4 εξηγεί γιατί χωρίς ταξινόμηση δεν ξέρεις ποιες πέντε θα πάρεις.

Η υπολογισμένη στήλη πήρε το όνομα `?column?`, που δεν λέει τίποτα. Με το `AS` της δίνεις **alias**, ένα όνομα που ισχύει μόνο για το αποτέλεσμα.

SQL

```
SELECT name, price * stock AS stock_value
FROM products
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                | stock_value
------------------------------------+-------------
 Βίος και πολιτεία του Αλέξη Ζορμπά |      709.80
 Το Κιβώτιο                         |      261.00
 Η Φόνισσα                          |      544.50
 Ματωμένα χώματα                    |        0.00
 Μαθαίνω Python                     |      800.00
(5 rows)
```

Για κείμενο ο τελεστής `||` ενώνει δύο τιμές. Έτσι φτιάχνεις το ονοματεπώνυμο ενός πελάτη από δύο στήλες.

SQL

```
SELECT first_name || ' ' || last_name AS full_name, email
FROM customers
LIMIT 4;
```

ΑΠΟΤΕΛΕΣΜΑ

```
      full_name       |        email
----------------------+---------------------
 Μαρία Παπαδοπούλου   | maria.p@example.gr
 Γιώργος Παπαδόπουλος | gpapad@example.gr
 Ελένη Νικολάου       | eleni.nik@example.gr
 Κώστας Δημητρίου     | kostas.d@example.gr
(4 rows)
```

Τα μονά εισαγωγικά ορίζουν **τιμή κειμένου** (string literal). Το `' '` είναι ένα κείμενο με ένα κενό. Τα διπλά εισαγωγικά ορίζουν **όνομα**, για παράδειγμα alias με κενά ή κεφαλαία.

SQL

```
SELECT name AS "Προϊόν", price AS "Τιμή €"
FROM products
LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
               Προϊόν               | Τιμή €
------------------------------------+--------
 Βίος και πολιτεία του Αλέξη Ζορμπά |  16.90
 Το Κιβώτιο                         |  14.50
 Η Φόνισσα                          |   9.90
(3 rows)
```

ΠΑΓΙΔΑ

Μονά και διπλά εισαγωγικά δεν είναι το ίδιο πράγμα στην SQL. Το `'Αθήνα'` είναι κείμενο. Το `"Αθήνα"` είναι όνομα στήλης ή πίνακα με αυτό το όνομα. Αν γράψεις το δεύτερο εκεί που εννοείς το πρώτο, η PostgreSQL θα ψάξει στήλη που δεν υπάρχει.

SQL

```
SELECT "Αθήνα";
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  column "Αθήνα" does not exist
LINE 1: SELECT "Αθήνα"
               ^
```

Η λέξη `AS` μπορεί να παραλειφθεί και θα τη δεις συχνά χωρίς αυτήν. Εδώ τη γράφουμε πάντα, επειδή κάνει το query πιο ευανάγνωστο.

SELECT χωρίς πίνακα

Το `FROM` δεν είναι υποχρεωτικό στην PostgreSQL. Ένα `SELECT` χωρίς πίνακα υπολογίζει εκφράσεις μία φορά και επιστρέφει μία γραμμή. Είναι ο πιο γρήγορος τρόπος να δοκιμάσεις μια συνάρτηση ή έναν τελεστή.

```sql
SELECT 7 / 2 AS division, 7 % 2 AS remainder, upper('postgres') AS shout;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 division | remainder |  shout
----------+-----------+----------
        3 |         1 | POSTGRES
(1 row)
```

Το `7 / 2` έδωσε 3 και όχι 3.5. Όταν και οι δύο αριθμοί είναι ακέραιοι, η διαίρεση είναι ακέραια. Θα το ξαναδούμε στο κεφάλαιο 5 μαζί με τους υπόλοιπους τύπους.

Πώς γράφεται η SQL

Οι λέξεις κλειδιά όπως `SELECT` και `FROM` δεν έχουν διάκριση πεζών και κεφαλαίων. Το `select name from products` είναι το ίδιο query. Το βιβλίο γράφει τις λέξεις κλειδιά με κεφαλαία για να ξεχωρίζουν από τα ονόματα.

Και τα ονόματα πινάκων και στηλών μετατρέπονται σε πεζά, εκτός αν είναι σε διπλά εισαγωγικά. Το `SELECT NAME FROM PRODUCTS` δουλεύει. Αν όμως κάποιος φτιάξει πίνακα με όνομα `"Products"`, θα πρέπει να γράφει πάντα `"Products"` με τα εισαγωγικά. Γι' αυτό τα ονόματα στην PostgreSQL γράφονται συνήθως με πεζά και κάτω παύλα, όπως `order_items`.

Τα κενά και οι αλλαγές γραμμής δεν έχουν σημασία. Ένα μεγάλο query το χωρίζεις σε γραμμές ανά clause. Τα σχόλια γράφονται με δύο παύλες μέχρι το τέλος της γραμμής ή ανάμεσα σε `/*` και `*/`.

```sql
-- Τα τρία πρώτα προϊόντα με την τιμή τους
SELECT name,
       price   /* σε ευρώ */
FROM products
LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                | price
------------------------------------+-------
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90
 Το Κιβώτιο                         | 14.50
 Η Φόνισσα                          |  9.90
(3 rows)
```

ΚΡΑΤΑ ΑΥΤΟ

Ένα `SELECT` παράγει έναν νέο πίνακα. Το `FROM` λέει από πού έρχονται οι γραμμές. Η λίστα του `SELECT` λέει ποιες στήλες θα έχει το αποτέλεσμα και υπολογίζεται μία φορά για κάθε γραμμή. Η σειρά των γραμμών δεν είναι εγγυημένη μέχρι να τη ζητήσεις.

Ασκήσεις

Κάτω από κάθε άσκηση φαίνεται το αποτέλεσμα που πρέπει να πάρεις. Αν διαφέρει μόνο η σειρά των γραμμών, το query σου είναι σωστό. Η σειρά είναι το θέμα του κεφαλαίου 4.

ΑΣΚΗΣΗ 1.1 Όλες οι κατηγορίες

Εμφάνισε όλες τις στήλες του πίνακα `categories`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |       name        | parent_id
----+-------------------+-----------
  1 | Βιβλία            |         ∅
  2 | Λογοτεχνία        |         1
  3 | Τεχνολογία        |         1
  4 | Προγραμματισμός   |         3
  5 | Βάσεις δεδομένων  |         3
  6 | Ηλεκτρονικά       |         ∅
  7 | Υπολογιστές       |         6
  8 | Laptops           |         7
  9 | Περιφερειακά      |         7
 10 | Ήχος              |         6
 11 | Σπίτι             |         ∅
 12 | Κουζίνα           |        11
 13 | Γραφείο           |        11
(13 rows)
```

ΑΣΚΗΣΗ 1.2 Κατάλογος προμηθευτών

Εμφάνισε το όνομα, το email και το τηλέφωνο κάθε προμηθευτή από τον πίνακα `suppliers`, με αυτή τη σειρά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
         name          |              email               |    phone
-----------------------+----------------------------------+--------------
 Βιβλιοδιανομή Αττικής | orders@vivliodianomi.example.gr | 2105550101
 Techno Import         | sales@technoimport.example.gr   | 2310555202
 Ήχος & Εικόνα ΑΕ      | ∅                                | 2105550303
 Κρητική Οικιακή       | info@kritiki.example.gr          | ∅
 Nordic Office         | hello@nordicoffice.example.com   | +46855500404
 Θεσσαλική Χαρτική     | xartika@example.gr               | 2410555606
(6 rows)
```

ΑΣΚΗΣΗ 1.3 Περιθώριο κέρδους

Για τα πέντε πρώτα προϊόντα εμφάνισε το `name`, την `price`, το `cost` και τη διαφορά τιμής και κόστους με όνομα `margin`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                name                | price | cost  | margin
------------------------------------+-------+-------+--------
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90 |  9.10 |   7.80
 Το Κιβώτιο                         | 14.50 |  7.80 |   6.70
 Η Φόνισσα                          |  9.90 |  4.20 |   5.70
 Ματωμένα χώματα                    | 15.20 |  8.00 |   7.20
 Μαθαίνω Python                     | 32.00 | 17.50 |  14.50
(5 rows)
```

ΑΣΚΗΣΗ 1.4 Ονοματεπώνυμο με κόμμα

Εμφάνισε για τους τέσσερις πρώτους πελάτες μία στήλη `customer` με τη μορφή `Επώνυμο, Όνομα` και δίπλα την ημερομηνία γέννησης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       customer        | birth_date
-----------------------+------------
 Παπαδοπούλου, Μαρία   | 1988-04-12
 Παπαδόπουλος, Γιώργος | 1979-09-30
 Νικολάου, Ελένη       | 1995-01-22
 Δημητρίου, Κώστας     | ∅
(4 rows)
```

ΑΣΚΗΣΗ 1.5 Ελληνικές επικεφαλίδες

Εμφάνισε από τον πίνακα `employees` το `full_name` με επικεφαλίδα `Ονοματεπώνυμο` και το `title` με επικεφαλίδα `Θέση`, για τους πέντε πρώτους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     Ονοματεπώνυμο     |        Θέση
-----------------------+--------------------
 Ελένη Μαυρίδου        | CEO
 Νίκος Αλεξίου         | CTO
 Σοφία Κωνσταντίνου    | Head of Sales
 Δημήτρης Λαμπρόπουλος | Operations Manager
 Κατερίνα Ζαφειρίου    | Backend Developer
(5 rows)
```

ΑΣΚΗΣΗ 1.6 Αριθμητική χωρίς πίνακα

Χωρίς `FROM`, υπολόγισε σε ένα query το `17 / 5`, το υπόλοιπο της ίδιας διαίρεσης και το `17.0 / 5`. Δώσε στις στήλες τα ονόματα `quotient`, `remainder` και `exact`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 quotient | remainder |       exact
----------+-----------+--------------------
        3 |         2 | 3.4000000000000000
(1 row)
```

ΑΣΚΗΣΗ 1.7 Πόσα χρήματα κάθονται στο ράφι

Για τα τρία πρώτα προϊόντα εμφάνισε το `sku`, το απόθεμα και την αξία του αποθέματος σε τιμή κόστους με όνομα `cost_value`. Στην ίδια γραμμή βάλε και την αξία σε τιμή πώλησης με όνομα `retail_value`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    sku     | stock | cost_value | retail_value
------------+-------+------------+--------------
 BK-LIT-001 |    42 |     382.20 |       709.80
 BK-LIT-002 |    18 |     140.40 |       261.00
 BK-LIT-003 |    55 |     231.00 |       544.50
(3 rows)
```

ΚΕΦΑΛΑΙΟ 2

WHERE και λογικές συνθήκες

Το `WHERE` κρατά μόνο τις γραμμές για τις οποίες μια συνθήκη είναι αληθής. Οι συνθήκες χτίζονται από συγκρίσεις και ενώνονται με `AND`, `OR` και `NOT`. Η προτεραιότητα αυτών των τελεστών είναι η πρώτη πραγματική παγίδα της SQL.

Συγκρίσεις

Το `WHERE` γράφεται μετά το `FROM`. Περιέχει μια έκφραση που για κάθε γραμμή βγαίνει αληθής ή ψευδής. Στο αποτέλεσμα μένουν μόνο οι γραμμές όπου είναι αληθής.

SQL

```
SELECT name, price
FROM products
WHERE price > 300;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name            |  price
----------------------------+---------
 Laptop Aero 14             | 1149.00
 Laptop Pro 16              | 2199.00
 Laptop Student 15          |  649.00
 Οθόνη 27" 4K               |  329.00
 Γραφείο ρυθμιζόμενου ύψους |  459.00
 Laptop Classic 13          |  799.00
(6 rows)
```

Οι τελεστές σύγκρισης είναι οι αναμενόμενοι. Το `=` σημαίνει ίσο και το `<>` διάφορο. Υπάρχουν επίσης τα `<`, `>`, `<=` και `>=`. Η PostgreSQL δέχεται και το `!=` ως συνώνυμο του `<>`. Ένα μόνο `=` είναι σύγκριση, όχι ανάθεση. Η SQL δεν έχει μεταβλητές μέσα σε ένα query.

Για να συγκρίνεις κείμενο γράφεις την τιμή σε μονά εισαγωγικά. Η σύγκριση κειμένου είναι ακριβής. Κεφαλαία, πεζά και τόνοι μετράνε.

SQL

```
SELECT id, name, region
FROM cities
WHERE region = 'Κρήτη';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |   name   | region
----+----------+--------
  4 | Ηράκλειο | Κρήτη
  8 | Χανιά    | Κρήτη
(2 rows)
```

SQL

```
SELECT id, name
FROM cities
WHERE name = 'ΑΘΗΝΑ' OR name = 'Αθηνα';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | name
----+------
(0 rows)
```

Κανένα από τα δύο δεν ταιριάζει με το `Αθήνα`, οπότε το αποτέλεσμα είναι άδειο. Ένα άδειο αποτέλεσμα δεν είναι σφάλμα. Είναι ένα σύνολο χωρίς γραμμές και το `psql` το δείχνει με τις επικεφαλίδες και `(0 rows)`.

Οι τελεστές `<` και `>` δουλεύουν και σε κείμενο. Συγκρίνουν αλφαβητικά, σύμφωνα με το **collation** της βάσης. Η βάση του βιβλίου χρησιμοποιεί ελληνικό collation, οπότε το `'Βόλος' < 'Γιάννενα'` είναι αληθές όπως θα περίμενες από λεξικό.

Η στήλη boolean ως συνθήκη

Μια στήλη τύπου `boolean` είναι ήδη συνθήκη. Δεν χρειάζεται να γράψεις `active = true`.

SQL

```
SELECT sku, name, stock
FROM products
WHERE NOT active;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  sku   |       name        | stock
--------+-------------------+-------
 LP-004 | Laptop Classic 13 |     0
(1 row)
```

Το `NOT` αντιστρέφει μια συνθήκη. Το `WHERE newsletter` κρατά όσους πελάτες έχουν εγγραφεί στο newsletter και το `WHERE NOT newsletter` όσους δεν έχουν.

AND, OR και παρενθέσεις

Το `AND` απαιτεί να ισχύουν και οι δύο συνθήκες. Το `OR` αρκείται σε μία. Οι πελάτες της Θεσσαλονίκης που έχουν εγγραφεί στο newsletter είναι αυτοί με `city_id` 2 και `newsletter` αληθές.

SQL

```
SELECT id, first_name, last_name
FROM customers
WHERE city_id = 2 AND newsletter;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name
----+------------+--------------
  6 | Νίκος      | Ιωάννου
 24 | Ζωή        | Παπαγεωργίου
 27 | Πέτρος     | Σαββίδης
(3 rows)
```

Όταν γράφεις και τα δύο στην ίδια συνθήκη, το `AND` εκτελείται πριν από το `OR`, όπως ο πολλαπλασιασμός πριν από την πρόσθεση. Ας πούμε ότι θέλεις τα προϊόντα των κατηγοριών 8 ή 9 που κοστίζουν κάτω από 100 ευρώ. Η πρώτη διατύπωση που έρχεται στο μυαλό είναι λάθος.

SQL

```
SELECT name, category_id, price
FROM products
WHERE category_id = 8 OR category_id = 9 AND price < 100;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name          | category_id |  price
-----------------------+-------------+---------
 Laptop Aero 14        |           8 | 1149.00
 Laptop Pro 16         |           8 | 2199.00
 Laptop Student 15     |           8 |  649.00
 Ασύρματο ποντίκι      |           9 |   24.90
 Μηχανικό πληκτρολόγιο |           9 |   89.00
 Webcam HD             |           9 |   59.00
 Laptop Classic 13     |           8 |  799.00
(7 rows)
```

Τα laptops της κατηγορίας 8 εμφανίζονται παρόλο που κοστίζουν πάνω από 100. Η PostgreSQL διάβασε τη συνθήκη ως `category_id = 8 OR (category_id = 9 AND price < 100)`. Οι παρενθέσεις λένε αυτό που εννοούσες.

SQL

```
SELECT name, category_id, price
FROM products
WHERE (category_id = 8 OR category_id = 9) AND price < 100;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name          | category_id | price
-----------------------+-------------+-------
 Ασύρματο ποντίκι      |           9 | 24.90
 Μηχανικό πληκτρολόγιο |           9 | 89.00
 Webcam HD             |           9 | 59.00
(3 rows)
```

ΠΑΓΙΔΑ

Όταν μια συνθήκη έχει και `AND` και `OR`, βάζε πάντα παρενθέσεις, ακόμη κι όταν η προτεραιότητα θα σου έδινε το σωστό. Όποιος διαβάσει το query μετά από έναν μήνα δεν θα χρειάζεται να θυμάται τον κανόνα.

IN και BETWEEN

Πολλές συγκρίσεις ισότητας με `OR` πάνω στην ίδια στήλη γράφονται πιο καθαρά με `IN`. Η λίστα μπαίνει σε παρενθέσεις.

SQL

```
SELECT id, customer_id, status
FROM orders
WHERE status IN ('cancelled', 'refunded') AND customer_id IN (1, 6);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  | customer_id |  status
-----+-------------+-----------
  12 |           1 | cancelled
  21 |           1 | cancelled
  25 |           6 | cancelled
  45 |           6 | cancelled
  55 |           6 | refunded
 121 |           6 | cancelled
(6 rows)
```

Το `NOT IN` κρατά όσες γραμμές δεν ταιριάζουν με καμία τιμή της λίστας. Έχει μια σοβαρή παγίδα όταν η λίστα περιέχει τιμή που λείπει. Θα τη δούμε στο κεφάλαιο 3 και ξανά στο κεφάλαιο 13.

Το `BETWEEN a AND b` σημαίνει `>= a AND <= b`. Περιλαμβάνει και τα δύο άκρα.

SQL

```
SELECT name, price
FROM products
WHERE price BETWEEN 30 AND 45;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name             | price
-----------------------------+-------
 Μαθαίνω Python              | 32.00
 Καθαρός κώδικας στην πράξη | 38.00
 PostgreSQL σε βάθος         | 45.00
 Σχεδίαση βάσεων δεδομένων  | 36.00
 Δίκτυα υπολογιστών          | 41.00
 Βραστήρας                   | 34.90
 Φωτιστικό γραφείου LED      | 42.00
(7 rows)
```

Το `PostgreSQL σε βάθος` με τιμή ακριβώς 45 είναι μέσα. Για αριθμούς και ημερομηνίες χωρίς ώρα αυτό είναι συνήθως ό,τι θέλεις. Για χρονικές στιγμές είναι παγίδα και το κεφάλαιο 6 δείχνει γιατί.

LIKE και ILIKE

Το `LIKE` συγκρίνει κείμενο με ένα μοτίβο. Το `%` ταιριάζει με οποιοδήποτε κομμάτι κειμένου, ακόμη και κενό. Το _ ταιριάζει με ακριβώς έναν χαρακτήρα. Όλοι οι άλλοι χαρακτήρες πρέπει να ταιριάζουν ακριβώς.

SQL

```
SELECT sku, name
FROM products
WHERE name LIKE 'Laptop%';
```

ΑΠΟΤΕΛΕΣΜΑ

```
  sku   |       name
--------+-------------------
 LP-001 | Laptop Aero 14
 LP-002 | Laptop Pro 16
 LP-003 | Laptop Student 15
 LP-004 | Laptop Classic 13
(4 rows)
```

Το `'Laptop%'` σημαίνει ότι το κείμενο ξεκινά με `Laptop`. Το `'%Laptop%'` θα σήμαινε ότι το περιέχει οπουδήποτε. Το _ είναι χρήσιμο όταν ξέρεις τη μορφή μιας τιμής. Όλα τα `sku` βιβλίων προγραμματισμού έχουν τη μορφή `BK-PRG-` και τρία ψηφία.

SQL

```
SELECT sku, name
FROM products
WHERE sku LIKE 'BK-PRG-___';
```

ΑΠΟΤΕΛΕΣΜΑ

```
    sku     |            name
------------+----------------------------
 BK-PRG-001 | Μαθαίνω Python
 BK-PRG-002 | Ruby από την αρχή
 BK-PRG-003 | Καθαρός κώδικας στην πράξη
(3 rows)
```

Το `LIKE` κάνει διάκριση πεζών και κεφαλαίων. Το `ILIKE` είναι η παραλλαγή της PostgreSQL που δεν κάνει. Δουλεύει και με ελληνικά, αλλά οι τόνοι εξακολουθούν να μετράνε.

SQL

```
SELECT id, name
FROM cities
WHERE name ILIKE 'αθ%' OR name ILIKE '%ανια';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | name
----+-------
  1 | Αθήνα
(1 row)
```

Το `Αθήνα` ταιριάζει. Το `Χανιά` όχι, επειδή το `ά` διαφέρει από το `α`. Το κεφάλαιο 28 δείχνει πώς αγνοείς τους τόνους με την επέκταση `unaccent` και πώς βρίσκεις λέξεις με ορθογραφικά λάθη με trigrams.

ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL

Το `ILIKE` δεν υπάρχει στο πρότυπο της SQL. Σε άλλες βάσεις θα δεις `lower(name) LIKE 'αθ%'`, που δουλεύει και στην PostgreSQL. Η διαφορά μετράει όταν ένα query πρέπει να τρέχει σε πολλές βάσεις.

Αν το κείμενο που ψάχνεις περιέχει το ίδιο `%` ή `_`, το προστατεύεις με backslash. Το `LIKE '%50\%%'` βρίσκει κείμενα που περιέχουν `50%`.

Το WHERE δεν βλέπει τα aliases

Ένα query δεν εκτελείται με τη σειρά που γράφεται. Λογικά, πρώτα διαλέγονται οι γραμμές του `FROM`, μετά φιλτράρονται από το `WHERE` και στο τέλος υπολογίζεται η λίστα του `SELECT`. Γι' αυτό το `WHERE` δεν ξέρει τα aliases του `SELECT`. Όταν εκτελείται, δεν έχουν υπολογιστεί ακόμη.

SQL

```
SELECT name, price * stock AS stock_value
FROM products
WHERE stock_value > 5000;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  column "stock_value" does not exist
LINE 3: WHERE stock_value > 5000
              ^
```

Γράφεις την έκφραση ξανά μέσα στο `WHERE`. Το κεφάλαιο 14 δείχνει έναν τρόπο να μην επαναλαμβάνεις μεγάλες εκφράσεις.

SQL

```
SELECT name, price * stock AS stock_value
FROM products
WHERE price * stock > 5000;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name        | stock_value
-------------------+-------------
 Laptop Aero 14    |     9192.00
 Laptop Pro 16     |     8796.00
 Laptop Student 15 |     9735.00
 Δωροκάρτα 50€     |    49950.00
(4 rows)
```

ΚΡΑΤΑ ΑΥΤΟ

Το `WHERE` κρατά τις γραμμές όπου η συνθήκη είναι αληθής. Το `AND` εκτελείται πριν από το `OR`, οπότε βάζεις παρενθέσεις. Το `IN` είναι συντομογραφία για πολλά `OR` και το `BETWEEN` περιλαμβάνει τα δύο άκρα. Το `LIKE` ταιριάζει μοτίβα με `%` και `_` και το `ILIKE` αγνοεί τα κεφαλαία.

Ασκήσεις

ΑΣΚΗΣΗ 2.1 Ακριβά προϊόντα

Εμφάνισε το όνομα και την τιμή των προϊόντων που κοστίζουν τουλάχιστον 459 ευρώ.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             |  price
-----------------------------+---------
 Laptop Aero 14              | 1149.00
 Laptop Pro 16               | 2199.00
 Laptop Student 15           |  649.00
 Γραφείο ρυθμιζόμενου ύψους |  459.00
 Laptop Classic 13           |  799.00
(5 rows)
```

ΑΣΚΗΣΗ 2.2 Εξαντλημένα αλλά ενεργά

Βρες τα προϊόντα που είναι ενεργά και δεν έχουν απόθεμα. Εμφάνισε `sku`, `name` και `stock`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    sku     |      name       | stock
------------+-----------------+-------
 BK-LIT-004 | Ματωμένα χώματα |     0
 PR-005     | Webcam HD       |     0
(2 rows)
```

ΑΣΚΗΣΗ 2.3 Πελάτες της Κρήτης

Οι πόλεις της Κρήτης στον πίνακα `cities` έχουν `id` 4 και 8. Βρες τους πελάτες που μένουν σε αυτές και δεν έχουν εγγραφεί στο newsletter. Εμφάνισε `id`, `first_name`, `last_name` και `city_id`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name   | city_id
----+------------+--------------+---------
  5 | Αναστασία  | Γεωργίου     |       4
 25 | Λευτέρης   | Φραγκιαδάκης |       8
 29 | Ηλίας      | Τζαννετάκης  |       4
(3 rows)
```

ΑΣΚΗΣΗ 2.4 Μεσαία τιμή

Εμφάνισε τα προϊόντα με τιμή από 50 έως 100 ευρώ, με τα δύο άκρα μέσα. Κράτα μόνο όσα ανήκουν στις κατηγορίες 9 ή 10.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
         name          | category_id | price
-----------------------+-------------+-------
 Μηχανικό πληκτρολόγιο |           9 | 89.00
 Webcam HD             |           9 | 59.00
 Ηχείο Bluetooth       |          10 | 79.00
(3 rows)
```

ΑΣΚΗΣΗ 2.5 Email εκτός .gr

Βρες τους προμηθευτές που έχουν email το οποίο δεν τελειώνει σε `.gr`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     name      |             email
---------------+--------------------------------
 Nordic Office | hello@nordicoffice.example.com
(1 row)
```

ΑΣΚΗΣΗ 2.6 Laptops που αξίζουν

Εμφάνισε όνομα, τιμή και απόθεμα για τα προϊόντα που το όνομά τους ξεκινά με `Laptop`, είναι ενεργά και κοστίζουν κάτω από 1500 ευρώ ή έχουν απόθεμα πάνω από 5.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       name        |  price  | stock
-------------------+---------+-------
 Laptop Aero 14    | 1149.00 |     8
 Laptop Student 15 |  649.00 |    15
(2 rows)
```

ΑΣΚΗΣΗ 2.7 Παραγγελίες που δεν παραδόθηκαν

Εμφάνισε τις παραγγελίες του πελάτη 6 που δεν είναι `delivered`. Δείξε `id`, `status` και `coupon_code`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  |  status   | coupon_code
-----+-----------+-------------
  25 | cancelled | WELCOME10
  45 | cancelled | ∅
  55 | refunded  | WELCOME10
 121 | cancelled | ∅
 143 | shipped   | ∅
(5 rows)
```

ΑΣΚΗΣΗ 2.8 Αρχή και τέλος ονόματος

Βρες τις πόλεις που το όνομά τους ξεκινά με `Ι` ή `Η` ή τελειώνει σε `ος`, χωρίς διάκριση πεζών και κεφαλαίων. Εμφάνισε όνομα και περιφέρεια.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   name   |    region
----------+--------------
 Ηράκλειο | Κρήτη
 Βόλος    | Θεσσαλία
 Ιωάννινα | Ήπειρος
 Ρόδος    | Νότιο Αιγαίο
(4 rows)
```

ΚΕΦΑΛΑΙΟ 3

NULL και λογική τριών τιμών

Το `NULL` σημαίνει ότι η τιμή δεν είναι γνωστή. Δεν είναι μηδέν ούτε κενό κείμενο. Κάθε σύγκριση με άγνωστη τιμή δίνει άγνωστο αποτέλεσμα και το `WHERE` πετά ό,τι δεν είναι σίγουρα αληθές.

Τι σημαίνει NULL

Ο πελάτης 5 δεν έχει δώσει email. Ο πελάτης 10 δεν έχει δηλώσει πόλη. Η δωροκάρτα δεν έχει κόστος αγοράς. Σε όλες αυτές τις θέσεις ο πίνακας κρατά `NULL`, την ειδική τιμή της SQL για κάτι που λείπει ή δεν είναι γνωστό.

SQL

```
SELECT id, first_name, email, phone, city_id
FROM customers
WHERE id IN (3, 5, 10, 14);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |         email          |   phone    | city_id
----+------------+------------------------+------------+---------
  3 | Ελένη      | eleni.nik@example.gr   | ∅          |       1
  5 | Αναστασία  | ∅                      | 6955667788 |       4
 10 | Χρήστος    | christos.m@example.gr  | ∅          |       ∅
 14 | Βασίλης    | ∅                      | ∅          |       2
(4 rows)
```

Το `NULL` δεν είναι το ίδιο με το κενό κείμενο `''` ή με το μηδέν. Ένα προϊόν με απόθεμα 0 έχει γνωστό απόθεμα που τυχαίνει να είναι μηδέν. Ένα email `''` είναι κείμενο χωρίς χαρακτήρες. Το `NULL` λέει ότι δεν ξέρουμε.

Συγκρίσεις με NULL

Αν δεν ξέρεις το email ενός πελάτη, δεν ξέρεις ούτε αν είναι ίσο με `maria.p@example.gr`. Η SQL ακολουθεί αυτή τη λογική κατά γράμμα. Κάθε σύγκριση με `NULL` δίνει `NULL`, ακόμη και η σύγκριση `NULL = NULL`.

SQL

```
SELECT 1 = NULL AS a, NULL = NULL AS b, NULL <> NULL AS c, 2 > NULL AS d;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 a | b | c | d
---+---+---+---
 ø | ø | ø | ø
(1 row)
```

Το αποτέλεσμα μιας σύγκρισης στην SQL έχει λοιπόν τρεις δυνατές τιμές. Αληθές, ψευδές και άγνωστο. Το `WHERE` κρατά μόνο τις γραμμές όπου η συνθήκη είναι αληθής. Όσες βγαίνουν ψευδείς ή άγνωστες φεύγουν. Γι' αυτό το παρακάτω query δεν επιστρέφει καμία γραμμή, παρόλο που υπάρχουν πελάτες χωρίς email.

SQL

```
SELECT id, first_name
FROM customers
WHERE email = NULL;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name
----+------------
(0 rows)
```

IS NULL και IS NOT NULL

Για να ρωτήσεις αν μια τιμή λείπει χρησιμοποιείς το `IS NULL`. Δεν είναι σύγκριση. Είναι έλεγχος που δίνει πάντα αληθές ή ψευδές.

SQL

```
SELECT id, first_name, last_name
FROM customers
WHERE email IS NULL;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | last_name
----+------------+-----------
  5 | Αναστασία  | Γεωργίου
 14 | Βασίλης    | Αντωνίου
(2 rows)
```

Το αντίθετο είναι το `IS NOT NULL`. Οι προμηθευτές για τους οποίους έχουμε και email και τηλέφωνο είναι αυτοί.

SQL

```
SELECT name, email, phone
FROM suppliers
WHERE email IS NOT NULL AND phone IS NOT NULL;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name          |              email               |    phone
-----------------------+----------------------------------+-------------
 Βιβλιοδιανομή Αττικής | orders@vivliodianomi.example.gr | 2105550101
 Techno Import         | sales@technoimport.example.gr   | 2310555202
 Nordic Office         | hello@nordicoffice.example.com  | +46855500404
 Θεσσαλική Χαρτική     | xartika@example.gr              | 2410555606
(4 rows)
```

Η παγίδα του διάφορου

Η πιο συχνή μορφή του προβλήματος εμφανίζεται με το `<>`. Θέλεις τους πελάτες που δεν μένουν στην Αθήνα.

SQL

```
SELECT id, first_name, city_id
FROM customers
WHERE city_id <> 1
  AND id < 12;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | city_id
----+------------+---------
  2 | Γιώργος    |       2
  4 | Κώστας     |       3
  5 | Αναστασία  |       4
  6 | Νίκος      |       2
  7 | Δήμητρα    |       5
  9 | Σοφία      |       6
 11 | Ιωάννα     |       7
(7 rows)
```

Ο πελάτης 10 λείπει. Δεν ξέρουμε πού μένει, οπότε το `city_id <> 1` είναι άγνωστο και το `WHERE` τον πετά. Αν η ερώτηση ήταν «ποιοι πελάτες δεν είναι σίγουρα Αθηναίοι», πρέπει να το πεις ρητά.

SQL

```
SELECT id, first_name, city_id
FROM customers
WHERE (city_id <> 1 OR city_id IS NULL)
  AND id < 12;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | city_id
----+------------+---------
  2 | Γιώργος    |       2
  4 | Κώστας     |       3
  5 | Αναστασία  |       4
  6 | Νίκος      |       2
  7 | Δήμητρα    |       5
  9 | Σοφία      |       6
 10 | Χρήστος    |       ∅
 11 | Ιωάννα     |       7
(8 rows)
```

Το ίδιο γράφεται πιο σύντομα με το `IS DISTINCT FROM`. Αντιμετωπίζει το `NULL` σαν κανονική τιμή. Δύο `NULL` θεωρούνται ίδια και ένα `NULL` είναι διαφορετικό από κάθε γνωστή τιμή.

SQL

```
SELECT id, first_name, city_id
FROM customers
WHERE city_id IS DISTINCT FROM 1
  AND id < 12;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | city_id
----+------------+---------
  2 | Γιώργος    |       2
  4 | Κώστας     |       3
  5 | Αναστασία  |       4
  6 | Νίκος      |       2
  7 | Δήμητρα    |       5
  9 | Σοφία      |       6
 10 | Χρήστος    |       ∅
 11 | Ιωάννα     |       7
(8 rows)
```

Το αντίθετό του είναι το `IS NOT DISTINCT FROM`, μια ισότητα που δεν χάνει τα `NULL`.

ΠΑΓΙΔΑ

Κάθε φορά που γράφεις `<>` ή `NOT` πάνω σε στήλη που επιτρέπει `NULL`, ρώτα τι πρέπει να γίνει με τις γραμμές όπου η τιμή λείπει. Η απάντηση σπάνια είναι «να εξαφανιστούν χωρίς να το καταλάβει κανείς».

Λογική τριών τιμών

Τα `AND`, `OR` και `NOT` ορίζονται και για το άγνωστο. Ο κανόνας είναι διαισθητικός αν σκεφτείς το άγνωστο ως «ίσως». Το `ψευδές AND ίσως` είναι σίγουρα ψευδές. Το `αληθές OR ίσως` είναι σίγουρα αληθές. Όλοι οι άλλοι συνδυασμοί με «ίσως» μένουν «ίσως».

Έκφραση	Αποτέλεσμα	Γιατί
`true AND NULL`	`NULL`	εξαρτάται από την άγνωστη τιμή
`false AND NULL`	`false`	ένα ψευδές αρκεί

Έκφραση	Αποτέλεσμα	Γιατί
`true OR NULL`	`true`	ένα αληθές αρκεί
`false OR NULL`	`NULL`	εξαρτάται από την άγνωστη τιμή
`NOT NULL`	`NULL`	το αντίθετο του αγνώστου είναι άγνωστο

Η συνέπεια για το `WHERE` είναι ότι το `NOT` δεν σώζει μια άγνωστη συνθήκη. Αν το `email LIKE '%.gr'` είναι άγνωστο για έναν πελάτη χωρίς email, τότε και το `NOT (email LIKE '%.gr')` είναι άγνωστο. Ο πελάτης λείπει και από τα δύο αποτελέσματα, όπως είδες στην άσκηση 2.5.

Το NOT IN και μια λίστα με NULL

Το `x NOT IN (1, 2, NULL)` σημαίνει `x <> 1 AND x <> 2 AND x <> NULL`. Ο τελευταίος όρος είναι πάντα άγνωστος, οπότε η συνθήκη δεν μπορεί ποτέ να βγει αληθής.

SQL

```
SELECT name
FROM cities
WHERE id NOT IN (1, 2, NULL);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 name
------
(0 rows)
```

Κανείς δεν γράφει `NULL` μέσα σε λίστα με το χέρι. Η λίστα όμως μπορεί να έρχεται από subquery πάνω σε στήλη με `NULL`. Τότε το query επιστρέφει σιωπηλά τίποτα. Το κεφάλαιο 13 δείχνει το `NOT EXISTS`, που δεν έχει αυτό το πρόβλημα.

Αριθμητική και κείμενο με NULL

Κάθε πράξη με `NULL` δίνει `NULL`. Το περιθώριο της δωροκάρτας δεν υπολογίζεται, επειδή δεν ξέρουμε το κόστος της.

SQL

```
SELECT name, price, cost, price - cost AS margin
FROM products
WHERE id IN (27, 28);
```

ΑΠΟΤΕΛΕΣΜΑ

```
          name          | price | cost  | margin
------------------------+-------+-------+--------
 Φωτιστικό γραφείου LED | 42.00 | 18.00 |  24.00
 Δωροκάρτα 50€          | 50.00 |     ∅ |      ∅
(2 rows)
```

Το ίδιο ισχύει για το `||`. Αν ένα κομμάτι λείπει, λείπει όλο το κείμενο. Οι συναρτήσεις `concat` και `concat_ws` αγνοούν τα `NULL`. Η δεύτερη παίρνει πρώτα το διαχωριστικό.

SQL

```
SELECT full_name,
       title || ' · ' || department AS with_pipes,
       concat_ws(' · ', title, department) AS with_concat_ws
FROM employees
WHERE id IN (1, 15);
```

ΑΠΟΤΕΛΕΣΜΑ

```
     full_name     |   with_pipes   |    with_concat_ws
-------------------+----------------+----------------------
 Ελένη Μαυρίδου    | CEO · Διοίκηση | CEO · Διοίκηση
 Λευτέρης Γεωργίου | ∅              | Εξωτερικός σύμβουλος
(2 rows)
```

COALESCE και NULLIF

Το `COALESCE` παίρνει μια λίστα εκφράσεων και επιστρέφει την πρώτη που δεν είναι `NULL`. Είναι ο συνηθισμένος τρόπος να δώσεις τιμή σε κάτι που λείπει.

SQL

```
SELECT first_name,
       COALESCE(email, phone, 'χωρίς στοιχεία') AS contact
FROM customers
WHERE id IN (1, 3, 5, 14);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name |       contact
------------+---------------------
 Μαρία      | maria.p@example.gr
 Ελένη      | eleni.nik@example.gr
 Αναστασία  | 6955667788
 Βασίλης    | χωρίς στοιχεία
(4 rows)
```

Ο πελάτης 14 δεν έχει ούτε email ούτε τηλέφωνο, οπότε φτάνουμε στην τελευταία τιμή. Όλες οι τιμές του `COALESCE` πρέπει να έχουν συμβατό τύπο. Δεν μπορείς να γράψεις `COALESCE(cost, 'άγνωστο')` σε αριθμητική στήλη.

Το `NULLIF(a, b)` κάνει το αντίθετο. Επιστρέφει `NULL` όταν το `a` είναι ίσο με το `b` και αλλιώς το `a`. Η κλασική χρήση του είναι να αποφύγεις διαίρεση με το μηδέν.

SQL

```
SELECT name, price / stock AS price_per_unit
FROM products
WHERE id IN (3, 4);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  division by zero
```

SQL

```
SELECT name, price / NULLIF(stock, 0) AS price_per_unit
FROM products
WHERE id IN (3, 4);
```

ΑΠΟΤΕΛΕΣΜΑ

```
      name       |     price_per_unit
-----------------+------------------------
 Η Φόνισσα       | 0.18000000000000000000
 Ματωμένα χώματα |                      ∅
(2 rows)
```

Το query δεν σπάει στη γραμμή με μηδενικό απόθεμα. Εκείνη η γραμμή παίρνει `NULL`, κάτι που είναι ειλικρινές. Ο λόγος δεν ορίζεται.

Πότε μια στήλη πρέπει να επιτρέπει NULL

Στο `\d cities` του κεφαλαίου 1 είδες το `not null` δίπλα σε τρεις στήλες. Όταν ορίζεις πίνακα αποφασίζεις ποιες στήλες μπορεί να λείπουν. Το email ενός πελάτη μπορεί να είναι άγνωστο. Η τιμή ενός προϊόντος όχι. Κάθε στήλη που επιτρέπει `NULL` χωρίς λόγο είναι μια ακόμη στήλη όπου θα πρέπει να σκεφτείς το `IS NULL` σε κάθε query. Το κεφάλαιο 17 επιστρέφει σε αυτή την απόφαση.

ΚΡΑΤΑ ΑΥΤΟ

Το `NULL` σημαίνει άγνωστο. Κάθε σύγκριση και κάθε πράξη με `NULL` δίνει `NULL` και το `WHERE` κρατά μόνο το αληθές. Ελέγχεις με `IS NULL` και `IS NOT NULL`, συγκρίνεις με `IS DISTINCT FROM` και αντικαθιστάς με `COALESCE`.

Ασκήσεις

ΑΣΚΗΣΗ 3.1 Πελάτες χωρίς τηλέφωνο

Εμφάνισε `id`, `first_name` και `last_name` των πελατών που δεν έχουν τηλέφωνο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name
----+------------+--------------
  3 | Ελένη      | Νικολάου
  7 | Δήμητρα    | Κωνσταντίνου
 10 | Χρήστος    | Μιχαηλίδης
 14 | Βασίλης    | Αντωνίου
 17 | Φωτεινή    | Λαμπράκη
 20 | Μιχάλης    | Τσακίρης
 24 | Ζωή        | Παπαγεωργίου
 28 | Δέσποινα   | Μαυρίδου
(8 rows)
```

ΑΣΚΗΣΗ 3.2 Κάποιο στοιχείο επικοινωνίας

Βρες τους πελάτες με δηλωμένη πόλη που έχουν τηλέφωνο αλλά όχι email ή email αλλά όχι τηλέφωνο. Εμφάνισε `id`, `email` και `phone`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |            email             |   phone
----+------------------------------+------------
  3 | eleni.nik@example.gr         | ∅
  5 | ∅                            | 6955667788
  7 | dimitra.k@example.gr         | ∅
 17 | foteini.l@example.gr         | ∅
 20 | michalis.ts@example.gr       | ∅
 24 | zoi.papageorgiou@example.gr | ∅
(6 rows)
```

ΑΣΚΗΣΗ 3.3 Εκτός Θεσσαλονίκης

Εμφάνισε `id` και `city_id` όλων των πελατών που δεν είναι σίγουρα από τη Θεσσαλονίκη (πόλη 2), μαζί με όσους δεν έχουν δηλώσει πόλη. Κράτα μόνο πελάτες με `id` από 1 έως 15.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | city_id
----+---------
  1 |       1
  3 |       1
  4 |       3
  5 |       4
  7 |       5
  8 |       1
  9 |       6
 10 |       ∅
 11 |       7
 12 |       1
 13 |       8
 15 |       9
(12 rows)
```

ΑΣΚΗΣΗ 3.4 Υπάλληλοι και μισθοί

Εμφάνισε το ονοματεπώνυμο κάθε υπαλλήλου που έχει `id` 11 ή μεγαλύτερο και τον μισθό του. Όπου ο μισθός λείπει, δείξε 0. Ονόμασε τη στήλη `salary_or_zero`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     full_name      | salary_or_zero
--------------------+----------------
 Βασιλική Ντόκου    |        1100.00
 Γιάννης Κορρές     |        1800.00
 Θανάσης Μπέης      |        1750.00
 Χριστίνα Αθανασίου |        1650.00
 Λευτέρης Γεωργίου  |              0
(5 rows)
```

ΑΣΚΗΣΗ 3.5 Τμήμα με προεπιλογή

Για τους υπαλλήλους 1, 11 και 15 εμφάνισε `full_name` και μια στήλη `label` της μορφής `Τίτλος (Τμήμα)`. Όταν το τμήμα λείπει, γράψε `χωρίς τμήμα` μέσα στην παρένθεση.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     full_name     |               label
-------------------+-------------------------------------
 Ελένη Μαυρίδου    | CEO (Διοίκηση)
 Βασιλική Ντόκου   | Sales Intern (Πωλήσεις)
 Λευτέρης Γεωργίου | Εξωτερικός σύμβουλος (χωρίς τμήμα)
(3 rows)
```

ΑΣΚΗΣΗ 3.6 Παραγγελίες χωρίς κουπόνι WELCOME10

Εμφάνισε `id` και `coupon_code` των παραγγελιών του πελάτη 6 που δεν χρησιμοποίησαν το κουπόνι `WELCOME10`, είτε επειδή είχαν άλλο κουπόνι είτε επειδή δεν είχαν κανένα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | coupon_code
-----+-------------
   2 | ∅
  39 | ∅
  42 | ∅
  45 | ∅
 102 | ∅
 121 | ∅
 143 | ∅
(7 rows)
```

ΑΣΚΗΣΗ 3.7 Περιθώριο χωρίς σφάλμα

Για τα προϊόντα 26 έως 28 υπολόγισε το περιθώριο ως ποσοστό της τιμής, δηλαδή `(price - cost) * 100 / price`, με όνομα `margin_pct`. Δίπλα βάλε στήλη `has_cost` που είναι αληθής όταν το κόστος είναι γνωστό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             |     margin_pct      | has_cost
-----------------------------+---------------------+----------
 Γραφείο ρυθμιζόμενου ύψους | 34.6405228758169935 | t
 Φωτιστικό γραφείου LED     | 57.1428571428571429 | t
 Δωροκάρτα 50€              |                   ∅ | f
(3 rows)
```

ΚΕΦΑΛΑΙΟ 4

ORDER BY, LIMIT και DISTINCT

Ένας πίνακας δεν έχει σειρά. Το `ORDER BY` τη δημιουργεί στο τέλος του query, το `LIMIT` κρατά τις πρώτες γραμμές και το `DISTINCT` πετά τα διπλότυπα. Όλα δουλεύουν πάνω στο αποτέλεσμα, όχι πάνω στον πίνακα.

Η λογική σειρά ενός query

Γράφεις `SELECT` πρώτα, αλλά λογικά εκτελείται αργότερα. Η σειρά με την οποία η PostgreSQL εφαρμόζει τα κομμάτια ενός query εξηγεί ποιο κομμάτι βλέπει ποια ονόματα. Στο κεφάλαιο 2 είδες ότι το `WHERE` δεν βλέπει τα aliases. Εδώ θα δεις ότι το `ORDER BY` τα βλέπει.

Σχήμα 4.1 · Η λογική σειρά ενός query με έναν πίνακα. Το κεφάλαιο 10 προσθέτει το GROUP BY και το HAVING ανάμεσα στο WHERE και στο SELECT.

Η σειρά είναι λογική, όχι πραγματική. Ο planner μπορεί να διαβάσει τις γραμμές ήδη ταξινομημένες από ένα index και να σταματήσει μόλις βρει όσες ζητά το `LIMIT`. Το αποτέλεσμα όμως είναι πάντα ίδιο με αυτό που θα έδινε η λογική σειρά.

ORDER BY

Το `ORDER BY` ταξινομεί το αποτέλεσμα κατά μία ή περισσότερες εκφράσεις. Η προεπιλογή είναι αύξουσα σειρά, `ASC`. Το `DESC` αντιστρέφει.

```sql
SELECT name, price
FROM products
WHERE category_id = 9
ORDER BY price DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name          | price
-----------------------+--------
 Οθόνη 27" 4K          | 329.00
 USB-C docking station | 139.00
 Μηχανικό πληκτρολόγιο |  89.00
 Webcam HD             |  59.00
 Ασύρματο ποντίκι      |  24.90
(5 rows)
```

Με πολλά κλειδιά, το δεύτερο κρίνει μόνο όταν το πρώτο είναι ίσο. Κάθε κλειδί έχει τη δική του κατεύθυνση. Εδώ οι πελάτες ταξινομούνται κατά πόλη και μέσα σε κάθε πόλη από τον πιο πρόσφατο.

SQL

```sql
SELECT city_id, first_name, created_at
FROM customers
WHERE city_id IN (1, 2)
ORDER BY city_id, created_at DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 city_id | first_name |       created_at
---------+------------+------------------------
       1 | Όλγα       | 2026-06-20 10:05:00+03
       1 | Στέλιος    | 2025-10-14 13:30:00+03
       1 | Αικατερίνη | 2025-08-03 11:20:00+03
       1 | Αλέξανδρος | 2025-04-02 09:40:00+03
       1 | Παναγιώτης | 2025-02-04 16:10:00+02
       1 | Ελένη      | 2024-12-01 09:05:00+02
       1 | Μαρία      | 2024-11-03 10:15:00+02
       2 | Πέτρος     | 2026-01-15 16:20:00+02
       2 | Ζωή        | 2025-11-02 10:50:00+02
       2 | Giorgos    | 2025-09-09 09:00:00+03
       2 | Βασίλης    | 2025-05-11 07:50:00+03
       2 | Νίκος      | 2025-01-09 13:45:00+02
       2 | Γιώργος    | 2024-11-20 18:40:00+02
(13 rows)
```

Το `ORDER BY` εκτελείται μετά το `SELECT`, οπότε βλέπει τα aliases. Μπορείς επίσης να ταξινομήσεις με έκφραση που δεν εμφανίζεται στο αποτέλεσμα.

SQL

```sql
SELECT name, price * stock AS stock_value
FROM products
ORDER BY stock_value DESC
LIMIT 4;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name        | stock_value
-------------------+-------------
 Δωροκάρτα 50€     |    49950.00
 Laptop Student 15 |     9735.00
 Laptop Aero 14    |     9192.00
 Laptop Pro 16     |     8796.00
(4 rows)
```

ΠΑΓΙΔΑ

Η PostgreSQL δέχεται και `ORDER BY 2`, με τον αριθμό της στήλης στη λίστα του `SELECT`. Είναι βολικό όταν πληκτρολογείς στο `psql`. Σε κώδικα που μένει, κάποιος θα προσθέσει μια στήλη στην αρχή της λίστας και η ταξινόμηση θα αλλάξει χωρίς κανένα μήνυμα.

Πού μπαίνουν τα NULL

Στην PostgreSQL το `NULL` ταξινομείται ως μεγαλύτερο από κάθε τιμή. Με `ASC` έρχεται στο τέλος και με `DESC` στην αρχή. Το δεύτερο σπάνια είναι αυτό που θέλεις. Οι νεότεροι πελάτες κατά ημερομηνία γέννησης δεν είναι αυτοί που δεν τη δήλωσαν.

SQL

```
SELECT first_name, birth_date
FROM customers
ORDER BY birth_date DESC
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name | birth_date
------------+------------
 Δέσποινα   | ∅
 Αλέξανδρος | ∅
 Κώστας     | ∅
 Μιχάλης    | ∅
 Αικατερίνη | 2003-05-05
(5 rows)
```

Το `NULLS LAST` και το `NULLS FIRST` ορίζουν ρητά τη θέση τους.

SQL

```
SELECT first_name, birth_date
FROM customers
ORDER BY birth_date DESC NULLS LAST
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name | birth_date
------------+------------
 Αικατερίνη | 2003-05-05
 Αναστασία  | 2001-07-08
 Ζωή        | 2000-11-30
 Σοφία      | 1999-12-02
 Φωτεινή    | 1998-03-17
(5 rows)
```

Ταξινόμηση κειμένου και collation

Η σειρά του κειμένου δεν είναι αντικειμενική. Εξαρτάται από τη γλώσσα και την ορίζει το **collation**. Η βάση του βιβλίου χρησιμοποιεί το ελληνικό collation της βιβλιοθήκης ICU. Τα τονισμένα γράμματα ταξινομούνται μαζί με τα άτονα, όπως σε ένα λεξικό.

SQL

```
SELECT first_name
FROM customers
WHERE first_name LIKE 'Α%' OR first_name LIKE 'Ά%' OR first_name LIKE 'Ό%'
ORDER BY first_name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name
------------
 Αικατερίνη
 Αλέξανδρος
 Αναστασία
 Άννα
 Όλγα
(5 rows)
```

Το collation `"C"` συγκρίνει τους κωδικούς των χαρακτήρων. Είναι γρήγορο, αλλά τα τονισμένα κεφαλαία έχουν μικρότερο κωδικό από τα άτονα, οπότε το `Άννα` και το `Όλγα` βγαίνουν πρώτα.

SQL

```
SELECT first_name
FROM customers
WHERE first_name LIKE 'Α%' OR first_name LIKE 'Ά%' OR first_name LIKE 'Ό%'
ORDER BY first_name COLLATE "C";
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name
------------
 Άννα
 Όλγα
 Αικατερίνη
 Αλέξανδρος
 Αναστασία
(5 rows)
```

ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL

Αν η βάση σου φτιάχτηκε χωρίς ελληνικό collation, η σειρά του κειμένου στα αποτελέσματα μπορεί να διαφέρει από του βιβλίου. Γι' αυτό το `setup.sh` φτιάχνει τη βάση με `--icu-locale=el-GR`. Το `COLLATE` σε ένα `ORDER BY` αλλάζει τη σειρά μόνο για εκείνο το query.

LIMIT, OFFSET και σταθερή σειρά

Το `LIMIT n` κρατά τις πρώτες `n` γραμμές του αποτελέσματος. Χωρίς `ORDER BY` δεν ορίζεται ποιες είναι οι πρώτες. Η PostgreSQL επιστρέφει όποιες βρει πρώτες. Αυτές μπορεί να αλλάξουν όταν αλλάξουν τα δεδομένα ή το plan. Στα προηγούμενα κεφάλαια τα αποτελέσματα έμοιαζαν σταθερά μόνο επειδή ο πίνακας γράφτηκε μία φορά και δεν άλλαξε.

Το `OFFSET k` παραλείπει τις πρώτες `k` γραμμές. Μαζί φτιάχνουν σελίδες. Η δεύτερη σελίδα με πέντε προϊόντα ανά σελίδα, από το φθηνότερο, είναι αυτή.

SQL

```
SELECT id, name, price
FROM products
ORDER BY price, id
LIMIT 5 OFFSET 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |            name             | price
----+-----------------------------+-------
  6 | Ruby από την αρχή           | 29.50
  5 | Μαθαίνω Python              | 32.00
 23 | Βραστήρας                   | 34.90
  9 | Σχεδίαση βάσεων δεδομένων   | 36.00
  7 | Καθαρός κώδικας στην πράξη | 38.00
(5 rows)
```

Το `id` στο τέλος του `ORDER BY` δεν είναι διακοσμητικό. Αν δύο προϊόντα είχαν την ίδια τιμή, η σειρά τους θα ήταν απροσδιόριστη και ένα από αυτά θα μπορούσε να εμφανιστεί σε δύο σελίδες και το άλλο σε καμία. Ένα κλειδί που είναι μοναδικό, όπως το primary key, κάνει τη σειρά **ντετερμινιστική**.

ΠΑΓΙΔΑ

Το `OFFSET` δεν παραλείπει γραμμές χωρίς κόστος. Για να δώσει τη σελίδα 1000, η βάση παράγει και πετά όλες τις προηγούμενες. Το κεφάλαιο 36 δείχνει το keyset pagination, που δεν έχει αυτό το πρόβλημα.

Το πρότυπο της SQL γράφει το ίδιο με `FETCH FIRST n ROWS ONLY`. Η μορφή αυτή έχει μια επιλογή που δεν έχει το `LIMIT`. Το `WITH TIES` κρατά και όσες γραμμές ισοβαθμούν με την τελευταία.

SQL

```
SELECT name, stock
FROM products
ORDER BY stock
FETCH FIRST 2 ROWS WITH TIES;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name        | stock
-------------------+-------
 Webcam HD         |     0
 Laptop Classic 13 |     0
 Ματωμένα χώματα   |     0
(3 rows)
```

Ζήτησες δύο γραμμές και πήρες τρεις, επειδή τρία προϊόντα έχουν μηδενικό απόθεμα. Με `LIMIT 2` ένα από αυτά θα έλειπε χωρίς να ξέρεις ποιο.

DISTINCT

Το `DISTINCT` αφαιρεί τις διπλές γραμμές από το αποτέλεσμα. Εφαρμόζεται σε ολόκληρη τη γραμμή, δηλαδή σε όλες τις στήλες του `SELECT` μαζί.

SQL

```
SELECT DISTINCT region
FROM cities
ORDER BY region;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       region
--------------------
 Αττική
 Δυτική Ελλάδα
 Δυτική Μακεδονία
 Ήπειρος
 Θεσσαλία
 Κεντρική Μακεδονία
 Κρήτη
 Νότιο Αιγαίο
 Πελοπόννησος
(9 rows)
```

SQL

```
SELECT DISTINCT status, coupon_code
FROM orders
WHERE customer_id = 6
ORDER BY status, coupon_code;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  status   | coupon_code
-----------+-------------
 cancelled | WELCOME10
 cancelled | ∅
 delivered | WELCOME10
 delivered | ∅
 refunded  | WELCOME10
 shipped   | ∅
(6 rows)
```

Για το `DISTINCT` δύο `NULL` θεωρούνται ίδια. Αυτό είναι μια από τις λίγες θέσεις όπου η SQL συγκρίνει `NULL` σαν να ήταν κανονική τιμή.

> **ΠΑΓΙΔΑ**
> Όταν ένα query επιστρέφει διπλές γραμμές που δεν περίμενες, το `DISTINCT` τις κρύβει αλλά δεν διορθώνει την αιτία. Συνήθως η αιτία είναι ένα join που πολλαπλασιάζει γραμμές. Το κεφάλαιο 11 επιστρέφει σε αυτό.

DISTINCT ON

Η PostgreSQL έχει μια επέκταση που λύνει ένα πολύ συχνό πρόβλημα, την πρώτη γραμμή από κάθε ομάδα. Το `DISTINCT ON (έκφραση)` κρατά μία γραμμή για κάθε διαφορετική τιμή της έκφρασης. Ποια γραμμή κρατά το ορίζει το `ORDER BY`, που πρέπει να ξεκινά με την ίδια έκφραση.

Το ακριβότερο προϊόν κάθε κατηγορίας είναι αυτό.

SQL

```
SELECT DISTINCT ON (category_id) category_id, name, price
FROM products
WHERE category_id IS NOT NULL
ORDER BY category_id, price DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 category_id |                name                |  price
-------------+------------------------------------+---------
           2 | Βίος και πολιτεία του Αλέξη Ζορμπά |   16.90
           3 | Δίκτυα υπολογιστών                 |   41.00
           4 | Καθαρός κώδικας στην πράξη         |   38.00
           5 | PostgreSQL σε βάθος                |   45.00
           8 | Laptop Pro 16                      | 2199.00
           9 | Οθόνη 27" 4K                       |  329.00
          10 | Πικάπ                              |  229.00
          12 | Καφετιέρα espresso                 |  249.00
          13 | Γραφείο ρυθμιζόμενου ύψους         |  459.00
(9 rows)
```

Το `ORDER BY category_id, price DESC` ταξινομεί κάθε ομάδα από την ακριβότερη τιμή και το `DISTINCT ON` κρατά την πρώτη γραμμή κάθε ομάδας. Αν δεν γράψεις το δεύτερο κλειδί, η PostgreSQL θα κρατήσει μια τυχαία γραμμή από κάθε κατηγορία.

> **ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL**
> Το `DISTINCT ON` δεν υπάρχει σε άλλες βάσεις. Το ίδιο πρόβλημα λύνεται παντού με window functions, όπως θα δεις στο κεφάλαιο 23.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το `ORDER BY` είναι ο μόνος τρόπος να ορίσεις σειρά και εκτελείται σχεδόν τελευταίο, οπότε βλέπει τα aliases. Βάλε πάντα ένα μοναδικό κλειδί στο τέλος όταν σελιδοποιείς. Το `DISTINCT` συγκρίνει ολόκληρες γραμμές και το `DISTINCT ON` κρατά την πρώτη γραμμή κάθε ομάδας.

Ασκήσεις

ΑΣΚΗΣΗ 4.1 Τα πέντε ακριβότερα

Εμφάνισε όνομα και τιμή για τα πέντε ακριβότερα ενεργά προϊόντα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             |  price
-----------------------------+---------
 Laptop Pro 16               | 2199.00
 Laptop Aero 14              | 1149.00
 Laptop Student 15           |  649.00
 Γραφείο ρυθμιζόμενου ύψους |  459.00
 Οθόνη 27" 4K                |  329.00
(5 rows)
```

ΑΣΚΗΣΗ 4.2 Κατάλογος πελατών

Εμφάνισε επώνυμο και όνομα των πελατών της Αθήνας (πόλη 1), ταξινομημένους κατά επώνυμο και μετά κατά όνομα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  last_name   | first_name
--------------+------------
 Βασιλείου    | Παναγιώτης
 Βλάχου       | Όλγα
 Ζαχαρίου     | Αικατερίνη
 Κουτσούκος   | Στέλιος
 Νικολάου     | Ελένη
 Οικονόμου    | Αλέξανδρος
 Παπαδοπούλου | Μαρία
(7 rows)
```

ΑΣΚΗΣΗ 4.3 Οι νεότεροι πελάτες

Εμφάνισε όνομα και ημερομηνία γέννησης των τεσσάρων νεότερων πελατών. Όσοι δεν έχουν δηλώσει ημερομηνία δεν πρέπει να εμφανίζονται.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 first_name | birth_date
------------+------------
 Αικατερίνη | 2003-05-05
 Αναστασία  | 2001-07-08
 Ζωή        | 2000-11-30
 Σοφία      | 1999-12-02
(4 rows)
```

ΑΣΚΗΣΗ 4.4 Τρίτη σελίδα

Ο κατάλογος εμφανίζει τα προϊόντα από το ακριβότερο στο φθηνότερο, τέσσερα ανά σελίδα. Εμφάνισε `id`, `name` και `price` της τρίτης σελίδας.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |         name          | price
----+-----------------------+--------
 30 | Πικάπ                 | 229.00
 19 | Ακουστικά ANC         | 199.00
 17 | USB-C docking station | 139.00
 21 | Μικρόφωνο USB         | 119.00
(4 rows)
```

ΑΣΚΗΣΗ 4.5 Περιφέρειες των πελατών

Εμφάνισε χωρίς επαναλήψεις τα ζεύγη `city_id` και `newsletter` των πελατών, ταξινομημένα κατά `city_id` με τα άγνωστα στην αρχή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 city_id | newsletter
---------+------------
       ∅ | f
       1 | f
       1 | t
       2 | f
       2 | t
       3 | f
       4 | f
       4 | t
       5 | f
       5 | t
       6 | t
       7 | f
       7 | t
       8 | f
 …
(17 rows)
```

ΑΣΚΗΣΗ 4.6 Ισοβαθμίες μισθών

Εμφάνισε όνομα και μισθό των τριών υπαλλήλων με τον χαμηλότερο γνωστό μισθό. Χρησιμοποίησε `FETCH FIRST` με τρόπο που δεν θα έκοβε ισοβαθμίες, αν υπήρχαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     full_name      | salary
--------------------+---------
 Βασιλική Ντόκου    | 1100.00
 Χριστίνα Αθανασίου | 1650.00
 Θανάσης Μπέης      | 1750.00
(3 rows)
```

ΑΣΚΗΣΗ 4.7 Η πιο πρόσφατη παραγγελία κάθε πελάτη

Για κάθε έναν από τους πελάτες 1 έως 5 εμφάνισε `customer_id`, το `id` και την ημερομηνία `created_at` της πιο πρόσφατης παραγγελίας του.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | id  |       created_at
-------------+-----+------------------------
           1 | 129 | 2026-05-26 09:44:00+03
           2 | 128 | 2026-05-25 17:04:00+03
           3 | 145 | 2026-06-14 22:04:00+03
           4 | 141 | 2026-06-09 18:41:00+03
           5 | 137 | 2026-06-06 22:00:00+03
(5 rows)
```

ΚΕΦΑΛΑΙΟ 5

Τύποι, εκφράσεις και CASE

Κάθε τιμή στην PostgreSQL έχει τύπο και ο τύπος αποφασίζει τι κάνουν οι τελεστές. Ο ίδιος τελεστής `/` κάνει ακέραια διαίρεση σε ακέραιους και ακριβή σε δεκαδικούς. Το `CASE` προσθέτει επιλογή μέσα σε μια έκφραση.

Ο τύπος μιας τιμής

Η συνάρτηση `pg_typeof` λέει τον τύπο οποιασδήποτε έκφρασης. Είναι το πρώτο εργαλείο όταν ένα αποτέλεσμα δεν είναι αυτό που περίμενες.

SQL

```
SELECT pg_typeof(42) AS a,
       pg_typeof(42.5) AS b,
       pg_typeof('42') AS c,
       pg_typeof(price) AS d,
       pg_typeof(tags) AS e
FROM products
WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    a    |    b    |    c    |    d    |   e
---------+---------+---------+---------+--------
 integer | numeric | unknown | numeric | text[]
(1 row)
```

Ο αριθμός `42` είναι `integer` και ο `42.5` είναι `numeric`. Το `'42'` είναι `unknown`, ένα κείμενο που η PostgreSQL δεν έχει αποφασίσει ακόμη πώς θα το διαβάσει. Θα το ερμηνεύσει ανάλογα με το πού το χρησιμοποιείς. Η στήλη `tags` είναι `text[]`, δηλαδή πίνακας από κείμενα. Θα τη δούμε στο κεφάλαιο 27.

Ακέραιοι, numeric και float

Οι αριθμητικοί τύποι που θα συναντάς είναι τρεις οικογένειες.

Τύπος	Τι κρατά	Πότε τον θέλεις
`integer`, `bigint`	ακέραιους 4 και 8 bytes	μετρήσεις, ποσότητες, κλειδιά
`numeric(p,s)`	ακριβείς δεκαδικούς	χρήματα και ό,τι απαιτεί ακρίβεια
`double precision`	δυαδική κινητή υποδιαστολή	μετρήσεις φυσικών μεγεθών, στατιστική

Η διαφορά ανάμεσα στο `numeric` και στο `double precision` φαίνεται αμέσως.

```sql
SELECT 0.1 + 0.2 = 0.3 AS numeric_equal,
       0.1::double precision + 0.2::double precision AS float_sum;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 numeric_equal |      float_sum
---------------+---------------------
 t             | 0.30000000000000004
(1 row)
```

Ο τύπος `double precision` δεν μπορεί να αναπαραστήσει το 0.1 ακριβώς. Για χρήματα αυτό είναι απαράδεκτο. Οι τιμές του καταστήματος είναι `numeric(10,2)`, δηλαδή μέχρι 10 ψηφία από τα οποία 2 μετά την υποδιαστολή.

Μετατροπή τύπων

Μετατρέπεις μια τιμή σε άλλο τύπο με `CAST(τιμή AS τύπος)` ή με τη συντομογραφία της PostgreSQL `τιμή::τύπος`. Θα δεις σχεδόν πάντα τη δεύτερη.

```sql
SELECT '42'::integer + 1 AS a,
       CAST('2026-06-30' AS date) AS b,
       7::numeric / 2 AS c,
       price::integer AS d
FROM products
WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 a  |     b      |         c          | d
----+------------+--------------------+----
 43 | 2026-06-30 | 3.5000000000000000 | 17
(1 row)
```

Το `price::integer` στρογγύλεψε το 16.90 σε 17. Η μετατροπή σε ακέραιο δεν κόβει, στρογγυλεύει. Αν η τιμή δεν μετατρέπεται, το query αποτυγχάνει ολόκληρο.

```sql
SELECT 'δεκαεπτά'::integer;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  invalid input syntax for type integer: "δεκαεπτά"
LINE 1: SELECT 'δεκαεπτά'::integer
               ^
```

ΠΑΓΙΔΑ

Η ακέραια διαίρεση είναι η πιο συχνή αιτία λάθους υπολογισμού. Το `quantity / 2` με `quantity` 3 δίνει 1. Αν θέλεις δεκαδικό αποτέλεσμα, μετατρέπεις τουλάχιστον έναν όρο σε `numeric` πριν διαιρέσεις.

Στρογγυλοποίηση

Η διαίρεση `numeric` δίνει πολλά δεκαδικά. Το `round(x, n)` στρογγυλεύει σε `n` δεκαδικά. Το `trunc` κόβει. Το `ceil` και το `floor` πάνε στον επόμενο και στον προηγούμενο ακέραιο.

SQL

```
SELECT 2::numeric / 3 AS raw,
       round(2::numeric / 3, 2) AS rounded,
       trunc(2::numeric / 3, 2) AS truncated,
       ceil(2.1) AS up,
       floor(2.9) AS down;
```

ΑΠΟΤΕΛΕΣΜΑ

```
          raw           | rounded | truncated | up | down
------------------------+---------+-----------+----+------
 0.6666666666666666667 |    0.67 |      0.66 |  3 |    2
(1 row)
```

Υπάρχει μια λεπτομέρεια που αξίζει να θυμάσαι. Το `round` σε `numeric` στρογγυλεύει το μισό προς τα πάνω, όπως στο σχολείο. Σε `double precision` ακολουθεί τον κανόνα του υλικού, που στρογγυλεύει το μισό στον πλησιέστερο ζυγό.

SQL

```
SELECT round(2.5) AS numeric_half, round(2.5::double precision) AS float_half;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 numeric_half | float_half
--------------+------------
            3 |          2
(1 row)
```

Συναρτήσεις κειμένου

Η PostgreSQL έχει δεκάδες συναρτήσεις για κείμενο. Αυτές θα τις χρησιμοποιείς συνεχώς.

SQL

```
SELECT email,
       length(email) AS len,
       upper(left(email, 5)) AS start,
       position('@' IN email) AS at_pos,
       split_part(email, '@', 2) AS domain,
       replace(email, 'example', 'shop') AS replaced
FROM customers
WHERE id IN (1, 21);
```

ΑΠΟΤΕΛΕΣΜΑ

```
              email               | len | start | at_pos |   domain   |            replaced
----------------------------------+-----+-------+--------+------------+--------------------------------
 maria.p@example.gr               |  18 | MARIA |      8 | example.gr | maria.p@shop.gr
 Giorgos.Papadopoulos@Example.gr  |  31 | GIORG |     21 | Example.gr | Giorgos.Papadopoulos@Example.gr
(2 rows)
```

Το `split_part(κείμενο, διαχωριστικό, n)` κόβει το κείμενο στο διαχωριστικό και επιστρέφει το κομμάτι `n`. Με αρνητικό `n` μετράει από το τέλος. Το `substring(email FROM 3 FOR 4)` επιστρέφει τέσσερις χαρακτήρες από τη θέση 3. Οι θέσεις στην SQL ξεκινούν από το 1.

Το `trim` αφαιρεί κενά από τις άκρες. Το `lpad` και το `rpad` συμπληρώνουν μέχρι ένα μήκος. Το `format` γεμίζει ένα πρότυπο με τιμές, όπως το `printf`.

SQL

```
SELECT format('%s · %s €', trim('  Laptop  '), 1149.00) AS label,
       lpad('42', 6, '0') AS padded,
       initcap('ΘΕΣΣΑΛΟΝΙΚΗ') AS title_case;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       label        | padded | title_case
--------------------+--------+------------
 Laptop · 1149.00 € | 000042 | Θεσσαλονικη
(1 row)
```

Το `initcap` κάνει κεφαλαίο το πρώτο γράμμα κάθε λέξης και πεζά τα υπόλοιπα. Δεν ξέρει όμως πού μπαίνει ο τόνος, οπότε το αποτέλεσμα μένει άτονο.

> **ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL**
> Η μετατροπή σε κεφαλαία ακολουθεί το collation. Με το ελληνικό collation της βάσης, το `upper('Ηράκλειο')` δίνει `ΗΡΑΚΛΕΙΟ` χωρίς τόνο, όπως γράφονται τα ελληνικά κεφαλαία. Σε βάση με collation `"C"` ή με το collation του λειτουργικού θα έβλεπες άλλο αποτέλεσμα.

Ο τελεστής `||` μετατρέπει αυτόματα σε κείμενο έναν μη κειμενικό όρο, οπότε το `'Τιμή ' || price` δουλεύει. Για ελεγχόμενη μορφή αριθμών υπάρχει το `to_char`. Το πρότυπο `FM9990.00` σημαίνει χωρίς κενά μπροστά, τουλάχιστον ένα ψηφίο πριν από την τελεία και ακριβώς δύο μετά.

```sql
SELECT name, to_char(price, 'FM9990.00') || ' €' AS shown
FROM products
WHERE id IN (3, 11);
```

ΑΠΟΤΕΛΕΣΜΑ

```
      name      |   shown
----------------+-----------
 Η Φόνισσα      | 9.90 €
 Laptop Aero 14 | 1149.00 €
(2 rows)
```

GREATEST και LEAST

Το `GREATEST` επιστρέφει τη μεγαλύτερη από μια λίστα τιμών και το `LEAST` τη μικρότερη. Αγνοούν τα `NULL`, εκτός αν όλες οι τιμές είναι `NULL`.

```sql
SELECT name, stock, GREATEST(stock - 10, 0) AS above_ten
FROM products
WHERE id IN (5, 6, 7);
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name             | stock | above_ten
-----------------------------+-------+-----------
 Μαθαίνω Python              |    25 |        15
 Ruby από την αρχή           |     9 |         0
 Καθαρός κώδικας στην πράξη |    14 |         4
(3 rows)
```

CASE

Το `CASE` είναι το `if` της SQL. Είναι έκφραση, οπότε δίνει τιμή και μπαίνει όπου μπαίνει μια έκφραση. Η γενική μορφή ελέγχει συνθήκες με τη σειρά και επιστρέφει την τιμή της πρώτης που είναι αληθής.

```sql
SELECT name, price,
       CASE
         WHEN price < 50 THEN 'οικονομικό'
         WHEN price < 300 THEN 'μεσαίο'
         ELSE 'ακριβό'
       END AS band
FROM products
WHERE category_id IN (9, 10)
ORDER BY price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name           | price  |    band
------------------------+--------+------------
 Ασύρματο ποντίκι       |  24.90 | οικονομικό
 Webcam HD              |  59.00 | μεσαίο
 Ηχείο Bluetooth        |  79.00 | μεσαίο
 Μηχανικό πληκτρολόγιο  |  89.00 | μεσαίο
 Μικρόφωνο USB          | 119.00 | μεσαίο
 USB-C docking station  | 139.00 | μεσαίο
 Ακουστικά ANC          | 199.00 | μεσαίο
 Πικάπ                  | 229.00 | μεσαίο
 Οθόνη 27" 4K           | 329.00 | ακριβό
(9 rows)
```

Η δεύτερη συνθήκη δεν χρειάζεται `price >= 50`. Αν η πρώτη ήταν αληθής, το `CASE` θα είχε ήδη σταματήσει. Αν καμία συνθήκη δεν είναι αληθής και δεν υπάρχει `ELSE`, το αποτέλεσμα είναι `NULL`.

Η απλή μορφή συγκρίνει μία έκφραση με λίστα τιμών. Είναι βολική για μεταφράσεις κωδικών.

SQL

```
SELECT id, status,
       CASE status
         WHEN 'pending' THEN 'σε αναμονή'
         WHEN 'paid' THEN 'πληρωμένη'
         WHEN 'shipped' THEN 'στον δρόμο'
         WHEN 'delivered' THEN 'παραδόθηκε'
         ELSE 'ακυρώθηκε'
       END AS status_gr
FROM orders
WHERE customer_id = 6
ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |  status   | status_gr
-----+-----------+------------
   2 | delivered | παραδόθηκε
  25 | cancelled | ακυρώθηκε
  26 | delivered | παραδόθηκε
  39 | delivered | παραδόθηκε
  42 | delivered | παραδόθηκε
  45 | cancelled | ακυρώθηκε
  55 | refunded  | ακυρώθηκε
 102 | delivered | παραδόθηκε
 121 | cancelled | ακυρώθηκε
 143 | shipped   | στον δρόμο
(10 rows)
```

Επειδή το `CASE` είναι έκφραση, μπορεί να μπει και στο `ORDER BY`. Έτσι ταξινομείς κατά μια σειρά που δεν είναι αλφαβητική, όπως τα στάδια μιας παραγγελίας.

SQL

```
SELECT id, status
FROM orders
WHERE customer_id = 6
ORDER BY CASE status
           WHEN 'pending' THEN 1
           WHEN 'paid' THEN 2
           WHEN 'shipped' THEN 3
           WHEN 'delivered' THEN 4
           ELSE 5
         END, id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |  status
-----+-----------
 143 | shipped
   2 | delivered
  26 | delivered
  39 | delivered
  42 | delivered
 102 | delivered
  25 | cancelled
  45 | cancelled
  55 | refunded
 121 | cancelled
(10 rows)
```

ΠΑΓΙΔΑ

Όλοι οι κλάδοι ενός `CASE` πρέπει να έχουν συμβατό τύπο. Ένα `CASE` που επιστρέφει αριθμό σε έναν κλάδο και κείμενο σε άλλον αποτυγχάνει. Αν θέλεις να δείξεις το κείμενο `άγνωστο` στη θέση ενός αριθμού, μετατρέπεις τον αριθμό σε κείμενο.

ΚΡΑΤΑ ΑΥΤΟ

Ο τύπος αποφασίζει τη συμπεριφορά. Χρήματα σε `numeric`, ποτέ σε `double precision`. Μετατρέπεις με `::`, στρογγυλεύεις με `round` και μορφοποιείς με `to_char` και `format`. Το `CASE` είναι έκφραση που επιλέγει τιμή και μπαίνει στο `SELECT`, στο `WHERE` και στο `ORDER BY`.

Ασκήσεις

ΑΣΚΗΣΗ 5.1 Τιμή με ΦΠΑ

Για τα βιβλία προγραμματισμού (το `sku` ξεκινά με `BK-PRG`) εμφάνισε όνομα, τιμή και τιμή με ΦΠΑ 24% στρογγυλεμένη σε δύο δεκαδικά, με όνομα `with_vat`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name            | price | with_vat
----------------------------+-------+----------
 Μαθαίνω Python             | 32.00 |    39.68
 Ruby από την αρχή          | 29.50 |    36.58
 Καθαρός κώδικας στην πράξη | 38.00 |    47.12
(3 rows)
```

ΑΣΚΗΣΗ 5.2 Domains των προμηθευτών

Εμφάνισε για κάθε προμηθευτή με email το όνομα και το domain του email σε πεζά, με όνομα `domain`. Ταξινόμησε κατά domain.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
         name           |          domain
------------------------+--------------------------
 Θεσσαλική Χαρτική      | example.gr
 Κρητική Οικιακή        | kritiki.example.gr
 Nordic Office          | nordicoffice.example.com
 Techno Import          | technoimport.example.gr
 Βιβλιοδιανομή Αττικής | vivliodianomi.example.gr
(5 rows)
```

ΑΣΚΗΣΗ 5.3 Κατηγορίες τιμής

Για τα βιβλία (το `sku` ξεκινά με `BK`) εμφάνισε όνομα, τιμή και στήλη `band` με τιμή `κάτω από 20` για τιμές μικρότερες του 20, `20 έως 40` για τιμές έως και 40 και `πάνω από 40` για τις υπόλοιπες. Ταξινόμησε από το φθηνότερο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                name                | price |    band
------------------------------------+-------+-------------
 Η Φόνισσα                          |  9.90 | κάτω από 20
 Το Κιβώτιο                         | 14.50 | κάτω από 20
 Ματωμένα χώματα                    | 15.20 | κάτω από 20
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90 | κάτω από 20
 Ruby από την αρχή                  | 29.50 | 20 έως 40
 Μαθαίνω Python                     | 32.00 | 20 έως 40
 Σχεδίαση βάσεων δεδομένων          | 36.00 | 20 έως 40
 Καθαρός κώδικας στην πράξη         | 38.00 | 20 έως 40
 Δίκτυα υπολογιστών                 | 41.00 | πάνω από 40
 PostgreSQL σε βάθος                | 45.00 | πάνω από 40
(10 rows)
```

ΑΣΚΗΣΗ 5.4 Αρχικά υπαλλήλων

Εμφάνισε για τους υπαλλήλους του τμήματος `Τεχνολογία` το ονοματεπώνυμο και τα αρχικά τους με τελείες, για παράδειγμα `Ν.Α.` για το `Νίκος Αλεξίου`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     full_name      | initials
--------------------+----------
 Νίκος Αλεξίου      | Ν.Α.
 Κατερίνα Ζαφειρίου | Κ.Ζ.
 Άγγελος Φωτίου     | Ά.Φ.
 Μαρίνα Σταθοπούλου | Μ.Σ.
 Πέτρος Χατζής      | Π.Χ.
(5 rows)
```

ΑΣΚΗΣΗ 5.5 Περιθώριο σε ποσοστό

Για τα προϊόντα της κατηγορίας 10 εμφάνισε όνομα και περιθώριο ως ποσοστό της τιμής, `(price - cost) * 100 / price`, στρογγυλεμένο σε ένα δεκαδικό. Ταξινόμησε από το μεγαλύτερο περιθώριο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      name       | margin_pct
-----------------+------------
 Ηχείο Bluetooth |       48.1
 Μικρόφωνο USB   |       46.2
 Ακουστικά ANC   |       39.7
 Πικάπ           |       38.9
(4 rows)
```

ΑΣΚΗΣΗ 5.6 Ανάλυση SKU

Για τα προϊόντα 11 έως 14 εμφάνισε το `sku`, το πρόθεμά του πριν από την πρώτη παύλα με όνομα `family` και το τελευταίο κομμάτι του ως ακέραιο με όνομα `seq`. Πρόσθεσε στήλη `next_seq` με τον επόμενο αριθμό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  sku   | family | seq | next_seq
--------+--------+-----+----------
 LP-001 | LP     |   1 |        2
 LP-002 | LP     |   2 |        3
 LP-003 | LP     |   3 |        4
 PR-001 | PR     |   1 |        2
(4 rows)
```

ΑΣΚΗΣΗ 5.7 Ετικέτα καταλόγου

Εμφάνισε για τα προϊόντα της κατηγορίας 12 μία στήλη `label` της μορφής `Βραστήρας · 34.90 € · σε απόθεμα`. Όταν το απόθεμα είναι κάτω από 10 γράψε `λίγα τεμάχια` αντί για `σε απόθεμα`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                    label
-----------------------------------------------
 Καφετιέρα espresso · 249.00 € · λίγα τεμάχια
 Βραστήρας · 34.90 € · σε απόθεμα
 Σετ μαχαιριών · 64.00 € · σε απόθεμα
(3 rows)
```

ΚΕΦΑΛΑΙΟ 6

Ημερομηνίες, ώρες και intervals

Η PostgreSQL ξεχωρίζει την ημερομηνία από τη χρονική στιγμή και τη διάρκεια. Το `timestamptz` κρατά ένα σημείο στον χρόνο και το εμφανίζει στη ζώνη ώρας της σύνδεσης. Τα διαστήματα γράφονται μισάνοιχτα και όχι με `BETWEEN`.

Οι τύποι του χρόνου

Τύπος	Παράδειγμα	Τι σημαίνει
`date`	`2026-06-30`	μια ημέρα του ημερολογίου, χωρίς ώρα
`time`	`14:30:00`	ώρα της ημέρας, χωρίς ημερομηνία
`timestamp`	`2026-06-30 14:30:00`	ημερομηνία και ώρα χωρίς ζώνη ώρας
`timestamptz`	`2026-06-30 14:30:00+03`	συγκεκριμένη στιγμή στον χρόνο
`interval`	`3 days 04:00:00`	διάρκεια

Στη βάση μας η `birth_date` και η `hired_on` είναι `date`, γιατί η ώρα δεν έχει νόημα. Η `created_at` των παραγγελιών είναι `timestamptz`, γιατί καταγράφει πότε ακριβώς συνέβη κάτι.

Οι τιμές γράφονται ως κείμενο σε μορφή ISO και μετατρέπονται στον τύπο. Μπροστά από το κείμενο μπορείς να γράψεις το όνομα του τύπου. Το `date '2026-06-30'` είναι το ίδιο με το `'2026-06-30'::date`.

SQL

```
SELECT date '2026-06-30' AS d,
       time '14:30' AS t,
       timestamptz '2026-06-30 14:30' AS ts,
       interval '3 days 4 hours' AS i;
```

ΑΠΟΤΕΛΕΣΜΑ

```
     d      |    t     |           ts           |        i
------------+----------+------------------------+-----------------
 2026-06-30 | 14:30:00 | 2026-06-30 14:30:00+03 | 3 days 04:00:00
(1 row)
```

timestamptz και ζώνες ώρας

Το όνομα `timestamp with time zone` παραπλανά. Ο τύπος δεν αποθηκεύει ζώνη ώρας. Αποθηκεύει μια στιγμή σε UTC. Όταν διαβάζεις την τιμή, η PostgreSQL τη μετατρέπει στη ζώνη της σύνδεσης, που

ορίζεται από τη ρύθμιση `timezone`. Η βάση του βιβλίου έχει `Europe/Athens`, οπότε οι στιγμές εμφανίζονται με `+02` τον χειμώνα και `+03` το καλοκαίρι.

Αν αλλάξεις τη ζώνη της σύνδεσης, αλλάζει η εμφάνιση αλλά όχι η στιγμή.

SQL

```
SELECT id, created_at FROM orders WHERE id = 1;
SET timezone = 'UTC';
SELECT id, created_at FROM orders WHERE id = 1;
RESET timezone;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |       created_at
----+------------------------
  1 | 2025-01-17 20:55:00+02
(1 row)

SET

 id |       created_at
----+------------------------
  1 | 2025-01-17 18:55:00+00
(1 row)

RESET
```

Το `AT TIME ZONE` μετατρέπει μια στιγμή σε ώρα τοίχου μιας συγκεκριμένης ζώνης, με τύπο `timestamp`. Είναι ο σωστός τρόπος να απαντήσεις «τι ώρα ήταν στη Νέα Υόρκη όταν έγινε η παραγγελία».

SQL

```
SELECT created_at,
       created_at AT TIME ZONE 'UTC' AS utc_wall,
       created_at AT TIME ZONE 'America/New_York' AS ny_wall
FROM orders
WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       created_at       |      utc_wall       |       ny_wall
------------------------+---------------------+---------------------
 2025-01-17 20:55:00+02 | 2025-01-17 18:55:00 | 2025-01-17 13:55:00
(1 row)
```

ΠΑΓΙΔΑ

Για στιγμές που συνέβησαν χρησιμοποίησε `timestamptz`. Ο απλός `timestamp` δεν ξέρει σε ποια ζώνη αναφέρεται η ώρα του. Δύο servers σε διαφορετικές ζώνες θα γράψουν το ίδιο γεγονός με διαφορετική τιμή και δεν θα μπορείς να τις συγκρίνεις.

Η τρέχουσα στιγμή

Το `now()` επιστρέφει την τρέχουσα στιγμή ως `timestamptz`. Το `current_date` επιστρέφει τη σημερινή ημερομηνία στη ζώνη της σύνδεσης. Μέσα σε ένα transaction το `now()` μένει σταθερό, ώστε όλες οι

εγγραφές ενός transaction να έχουν την ίδια χρονοσφραγίδα. Αν θέλεις την πραγματική ώρα την ώρα που εκτελείται κάθε εντολή, υπάρχει το `clock_timestamp()`.

Τα δεδομένα του βιβλίου σταματούν στις 30 Ιουνίου 2026. Για να βγαίνουν ίδια αποτελέσματα όποτε κι αν τρέξεις τα queries, τα παραδείγματα γράφουν αυτή την ημερομηνία ρητά αντί για `current_date`.

Αριθμητική με ημερομηνίες

Σε μια `date` προσθέτεις ακέραιο αριθμό ημερών. Η διαφορά δύο `date` είναι ακέραιος αριθμός ημερών. Η διαφορά δύο `timestamptz` είναι `interval`.

```
SQL

SELECT date '2026-06-30' + 7 AS next_week,
       date '2026-06-30' - date '2026-01-01' AS days_this_year,
       shipped_at - created_at AS shipping_time
FROM orders
WHERE id = 2;
```

```
ΑΠΟΤΕΛΕΣΜΑ

 next_week  | days_this_year |  shipping_time
------------+----------------+-----------------
 2026-07-07 |            180 | 4 days 03:00:00
(1 row)
```

Με `interval` προσθέτεις μήνες και έτη. Ένας μήνας δεν έχει σταθερή διάρκεια, οπότε η PostgreSQL προσθέτει ημερολογιακό μήνα. Όταν η ημέρα δεν υπάρχει στον επόμενο μήνα, κρατά την τελευταία ημέρα του.

```
SQL

SELECT date '2026-01-31' + interval '1 month' AS feb,
       date '2026-03-31' - interval '1 month' AS also_feb,
       date '2024-02-29' + interval '1 year' AS leap;
```

```
ΑΠΟΤΕΛΕΣΜΑ

         feb         |      also_feb       |        leap
---------------------+---------------------+---------------------
 2026-02-28 00:00:00 | 2026-02-28 00:00:00 | 2025-02-28 00:00:00
(1 row)
```

Το ίδιο ισχύει για τις ημέρες και την αλλαγή ώρας. Στις 29 Μαρτίου 2026 τα ρολόγια της Ελλάδας πήγαν από τις 03:00 στις 04:00. Ένα `interval '1 day'` προσθέτει ημερολογιακή ημέρα και κρατά την ώρα τοίχου. Ένα `interval '24 hours'` προσθέτει 24 πραγματικές ώρες.

```
SQL

SELECT timestamptz '2026-03-28 12:00' + interval '1 day' AS plus_day,
       timestamptz '2026-03-28 12:00' + interval '24 hours' AS plus_24h;
```

ΑΠΟΤΕΛΕΣΜΑ

```
        plus_day         |        plus_24h
-------------------------+------------------------
 2026-03-29 12:00:00+03 | 2026-03-29 13:00:00+03
(1 row)
```

Το `age(a, b)` δίνει τη διαφορά ως έτη, μήνες και ημέρες, κάτι βολικό για ηλικίες.

SQL

```
SELECT first_name, birth_date,
       age(date '2026-06-30', birth_date) AS age
FROM customers
WHERE id IN (1, 11);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name | birth_date |           age
------------+------------+--------------------------
 Μαρία      | 1988-04-12 | 38 years 2 mons 18 days
 Ιωάννα     | 1992-02-29 | 34 years 4 mons 1 day
(2 rows)
```

Κομμάτια μιας ημερομηνίας

Το `extract(πεδίο FROM τιμή)` βγάζει ένα κομμάτι ως αριθμό. Τα πεδία που θα χρειαστείς πιο συχνά είναι τα `year`, `month`, `day`, `hour`, `dow` και `isodow`. Στο `isodow` η Δευτέρα είναι 1 και η Κυριακή 7. Στο `dow` η Κυριακή είναι 0.

SQL

```
SELECT id, created_at,
       extract(year FROM created_at) AS y,
       extract(month FROM created_at) AS m,
       extract(isodow FROM created_at) AS weekday,
       extract(hour FROM created_at) AS h
FROM orders
WHERE id IN (1, 2, 3);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |       created_at       |  y   | m | weekday | h
----+------------------------+------+---+---------+----
  1 | 2025-01-17 20:55:00+02 | 2025 | 1 |       5 | 20
  2 | 2025-01-19 18:59:00+02 | 2025 | 1 |       7 | 18
  3 | 2025-01-28 16:54:00+02 | 2025 | 1 |       2 | 16
(3 rows)
```

Το `date_trunc('μονάδα', τιμή)` κόβει μια στιγμή στην αρχή της μονάδας. Είναι το βασικό εργαλείο για να ομαδοποιείς ανά ημέρα, εβδομάδα ή μήνα, όπως θα κάνουμε από το κεφάλαιο 10. Η εβδομάδα ξεκινά Δευτέρα.

```sql
SELECT created_at,
       date_trunc('month', created_at) AS month_start,
       date_trunc('week', created_at)::date AS week_start
FROM orders
WHERE id = 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       created_at       |      month_start       | week_start
------------------------+------------------------+------------
 2025-01-28 16:54:00+02 | 2025-01-01 00:00:00+02 | 2025-01-27
(1 row)
```

Για εμφάνιση υπάρχει το `to_char` με πρότυπο. Το `YYYY` είναι το έτος, το `MM` ο μήνας, το `DD` η ημέρα, το `HH24:MI` η ώρα.

```sql
SELECT to_char(created_at, 'YYYY-MM') AS month,
       to_char(created_at, 'DD/MM/YYYY HH24:MI') AS shown
FROM orders
WHERE id = 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  month  |      shown
---------+------------------
 2025-01 | 28/01/2025 16:54
(1 row)
```

Διαστήματα χρόνου

Θέλεις τις παραγγελίες του Μαρτίου 2026. Η λύση που φαίνεται φυσική είναι λάθος.

```sql
SELECT id, created_at
FROM orders
WHERE created_at BETWEEN '2026-03-01' AND '2026-03-31'
ORDER BY created_at DESC
LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |       created_at
-----+------------------------
 103 | 2026-03-25 19:45:00+02
 102 | 2026-03-25 14:10:00+02
 101 | 2026-03-23 19:53:00+02
(3 rows)
```

Το `'2026-03-31'` μετατρέπεται σε `2026-03-31 00:00`, οπότε κάθε παραγγελία της 31ης Μαρτίου μετά τα μεσάνυχτα λείπει. Η διόρθωση με `'2026-03-31 23:59:59'` αφήνει κενό το τελευταίο δευτερόλεπτο. Το σωστό είναι ένα **μισάνοιχτο διάστημα**. Κλειστό στην αρχή, ανοιχτό στο τέλος.

SQL

```
SELECT id, created_at
FROM orders
WHERE created_at >= '2026-03-01' AND created_at < '2026-04-01'
ORDER BY created_at DESC
LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |       created_at
-----+------------------------
 104 | 2026-03-31 11:30:00+03
 103 | 2026-03-25 19:45:00+02
 102 | 2026-03-25 14:10:00+02
(3 rows)
```

Τα μισάνοιχτα διαστήματα έχουν και ένα δεύτερο πλεονέκτημα. Διαδοχικά διαστήματα κολλάνε το ένα στο άλλο χωρίς κενά και χωρίς επικάλυψη, αφού το τέλος του ενός είναι η αρχή του επόμενου.

ΠΑΓΙΔΑ

Θα δεις συχνά το `WHERE created_at::date = '2026-03-31'`. Δίνει σωστό αποτέλεσμα, αλλά μετατρέπει κάθε γραμμή πριν από τη σύγκριση. Ένα index πάνω στη `created_at` δεν μπορεί να βοηθήσει. Το κεφάλαιο 30 εξηγεί γιατί η σύγκριση με το μισάνοιχτο διάστημα είναι αυτή που χρησιμοποιεί index.

Σειρές ημερομηνιών

Το `generate_series(αρχή, τέλος, βήμα)` παράγει μια σειρά τιμών. Με ημερομηνίες και `interval` παράγει ημερολόγιο. Εμφανίζεται στο `FROM` σαν να ήταν πίνακας.

SQL

```
SELECT d::date AS day, to_char(d, 'Dy') AS weekday
FROM generate_series(date '2026-06-01', date '2026-06-07', interval '1 day') AS d;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    day     | weekday
------------+---------
 2026-06-01 | Mon
 2026-06-02 | Tue
 2026-06-03 | Wed
 2026-06-04 | Thu
 2026-06-05 | Fri
 2026-06-06 | Sat
 2026-06-07 | Sun
(7 rows)
```

Το `AS d` ονομάζει τη στήλη που παράγεται. Τα ονόματα των ημερών βγαίνουν στα αγγλικά. Το `to_char` με `TMDy` θα τα έγραφε στη γλώσσα που ορίζει η ρύθμιση `lc_time` του server. Στο κεφάλαιο 25 θα χρησιμοποιήσουμε το `generate_series` για να συμπληρώσουμε ημέρες χωρίς παραγγελίες.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Χρησιμοποίησε `date` για ημέρες και `timestamptz` για στιγμές. Η διαφορά στιγμών είναι `interval`. Κόβεις με `date_trunc`, βγάζεις κομμάτια με `extract` και φιλτράρεις με μισάνοιχτα διαστήματα `>= αρχή AND < τέλος`.

Ασκήσεις

Στις ασκήσεις η «σήμερα» είναι η 30ή Ιουνίου 2026.

ΑΣΚΗΣΗ 6.1 Παραγγελίες του Φεβρουαρίου

Εμφάνισε `id`, `customer_id` και `created_at` των παραγγελιών του Φεβρουαρίου 2026, από την πιο πρόσφατη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | customer_id |       created_at
----+-------------+------------------------
 92 |          11 | 2026-02-23 21:54:00+02
 91 |           1 | 2026-02-23 21:07:00+02
 90 |           5 | 2026-02-23 14:48:00+02
 89 |           2 | 2026-02-23 09:41:00+02
 88 |           9 | 2026-02-22 12:45:00+02
 87 |           5 | 2026-02-19 14:30:00+02
 86 |          19 | 2026-02-14 12:23:00+02
 85 |          28 | 2026-02-14 11:22:00+02
 84 |           9 | 2026-02-12 16:22:00+02
 83 |           5 | 2026-02-05 20:54:00+02
 82 |          13 | 2026-02-04 09:09:00+02
 81 |          21 | 2026-02-02 08:04:00+02
(12 rows)
```

ΑΣΚΗΣΗ 6.2 Αργές αποστολές

Βρες τις παραγγελίες του 2026 που χρειάστηκαν πάνω από τέσσερις ημέρες για να αποσταλούν. Εμφάνισε `id`, την ημερομηνία παραγγελίας ως `date` και τον χρόνο αποστολής με όνομα `shipping_time`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | ordered_on |  shipping_time
-----+------------+-----------------
  76 | 2026-01-10 | 4 days 04:00:00
  79 | 2026-01-26 | 4 days 01:00:00
  82 | 2026-02-04 | 4 days 06:00:00
  85 | 2026-02-14 | 4 days 04:00:00
  90 | 2026-02-23 | 4 days 06:00:00
  93 | 2026-03-01 | 4 days 01:00:00
  94 | 2026-03-04 | 4 days 02:00:00
  95 | 2026-03-06 | 4 days 01:00:00
  98 | 2026-03-16 | 4 days 01:00:00
 102 | 2026-03-25 | 4 days 01:00:00
 104 | 2026-03-31 | 4 days 03:00:00
 112 | 2026-04-23 | 4 days 05:00:00
 132 | 2026-05-30 | 4 days 01:00:00
 136 | 2026-06-05 | 4 days 05:00:00
 …
(21 rows)
```

ΑΣΚΗΣΗ 6.3 Παραγγελίες του Σαββατοκύριακου

Εμφάνισε `id` και `created_at` των παραγγελιών του Μαΐου 2026 που έγιναν Σάββατο ή Κυριακή, με όνομα ημέρας στα αγγλικά σε στήλη `day`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  |       created_at       |    day
-----+------------------------+-----------
 117 | 2026-05-09 15:51:00+03 | Saturday
 121 | 2026-05-17 15:39:00+03 | Sunday
 125 | 2026-05-23 23:07:00+03 | Saturday
 126 | 2026-05-24 18:43:00+03 | Sunday
 132 | 2026-05-30 16:20:00+03 | Saturday
 133 | 2026-05-31 19:38:00+03 | Sunday
(6 rows)
```

ΑΣΚΗΣΗ 6.4 Ηλικίες

Για τους πελάτες που γεννήθηκαν πριν από το 1980 εμφάνισε όνομα, ημερομηνία γέννησης και την ηλικία τους σε ολόκληρα έτη στις 30 Ιουνίου 2026, με όνομα `years`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 first_name | birth_date | years
------------+------------+-------
 Βασίλης    | 1965-10-10 |    60
 Παναγιώτης | 1972-05-27 |    54
 Λευτέρης   | 1975-04-04 |    51
 Σπύρος     | 1976-12-24 |    49
 Γιώργος    | 1979-09-30 |    46
 Giorgos    | 1979-09-30 |    46
(6 rows)
```

ΑΣΚΗΣΗ 6.5 Μήνας και εβδομάδα

Για τις παραγγελίες του πελάτη 13 εμφάνισε `id`, τον μήνα σε μορφή `YYYY-MM` με όνομα `month` και την ημερομηνία της Δευτέρας της εβδομάδας με όνομα `week_start`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |  month  | week_start
----+---------+------------
 20 | 2025-05 | 2025-04-28
 35 | 2025-09 | 2025-09-01
 40 | 2025-09 | 2025-09-15
 41 | 2025-09 | 2025-09-15
 82 | 2026-02 | 2026-02-02
(5 rows)
```

ΑΣΚΗΣΗ 6.6 Οι Δευτέρες του Ιουνίου

Εμφάνισε με `generate_series` όλες τις Δευτέρες του Ιουνίου 2026 ως `date`, σε στήλη `monday`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   monday
------------
 2026-06-01
 2026-06-08
 2026-06-15
 2026-06-22
 2026-06-29
(5 rows)
```

ΑΣΚΗΣΗ 6.7 Η ίδια στιγμή σε τρεις πόλεις

Για την παραγγελία 100 εμφάνισε το `created_at` και την ώρα τοίχου της ίδιας στιγμής στο Λονδίνο και στο Τόκιο, σε στήλες `london` και `tokyo`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       created_at       |       london        |        tokyo
------------------------+---------------------+---------------------
 2026-03-22 13:04:00+02 | 2026-03-22 11:04:00 | 2026-03-22 20:04:00
(1 row)
```

ΑΣΚΗΣΗ 6.8 Πρόσφατες προσλήψεις

Εμφάνισε όνομα και ημερομηνία πρόσληψης των υπαλλήλων που προσλήφθηκαν τα δύο τελευταία χρόνια πριν από τις 30 Ιουνίου 2026, μαζί με τις ημέρες που έχουν περάσει από την πρόσληψη, σε στήλη `days`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     full_name     |  hired_on  | days
-------------------+------------+------
 Πέτρος Χατζής     | 2024-09-02 |  666
 Λευτέρης Γεωργίου | 2025-01-07 |  539
 Βασιλική Ντόκου   | 2025-10-01 |  272
(3 rows)
```

ΜΕΡΟΣ ΙΙ

Πολλοί πίνακες και συνόψεις

Τα δεδομένα μιας πραγματικής βάσης είναι μοιρασμένα σε πολλούς πίνακες. Εδώ τους ενώνουμε με joins, τους συνοψίζουμε με GROUP BY και ρωτάμε τον ένα με βάση τον άλλο με subqueries.

ΚΕΦΑΛΑΙΟ 7

Κλειδιά και INNER JOIN

Ένα foreign key λέει ότι μια στήλη δείχνει σε γραμμή άλλου πίνακα. Το `JOIN` ακολουθεί αυτή τη σχέση και φέρνει στην ίδια γραμμή του αποτελέσματος στήλες από δύο πίνακες. Το `INNER JOIN` κρατά μόνο τα ζεύγη που ταιριάζουν.

Γιατί τα δεδομένα είναι σε πολλούς πίνακες

Ο πίνακας `customers` δεν έχει στήλη με το όνομα της πόλης. Έχει τη `city_id`, έναν αριθμό που δείχνει σε γραμμή του πίνακα `cities`. Το όνομα της Θεσσαλονίκης γράφεται μία φορά, στη γραμμή 2 του `cities`. Όλοι οι πελάτες της Θεσσαλονίκης δείχνουν σε αυτήν.

Αν το όνομα ήταν γραμμένο σε κάθε πελάτη, μια αλλαγή θα έπρεπε να γίνει σε πολλές γραμμές. Μια ορθογραφική παραλλαγή σε μία από αυτές θα έφτιαχνε μια δεύτερη Θεσσαλονίκη. Ο διαχωρισμός κρατά κάθε γεγονός σε ένα σημείο. Το κεφάλαιο 18 επιστρέφει σε αυτή την ιδέα ως **κανονικοποίηση**.

Το `id` του `cities` είναι **primary key**. Είναι μοναδικό και δεν είναι ποτέ `NULL`, οπότε αναγνωρίζει μία γραμμή. Η `customers.city_id` είναι **foreign key**. Η βάση εγγυάται ότι κάθε τιμή της είτε υπάρχει στο `cities.id` είτε είναι `NULL`. Δεν μπορείς να βάλεις πελάτη στην πόλη 99 αν δεν υπάρχει πόλη 99.

Το πρώτο JOIN

Το `JOIN` γράφεται στο `FROM`. Δίπλα του δηλώνεις τον δεύτερο πίνακα και μετά το `ON` τη συνθήκη που ταιριάζει γραμμές.

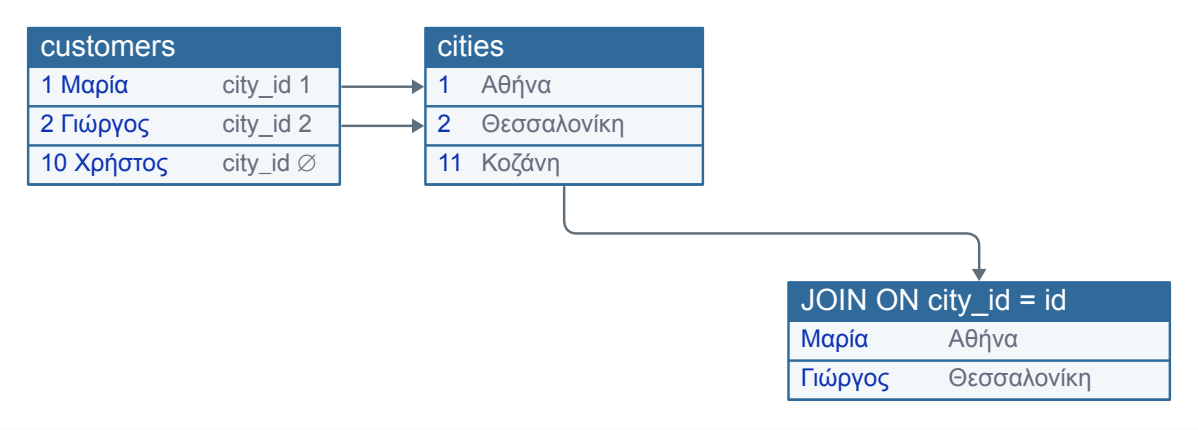
SQL

```
SELECT customers.first_name, customers.last_name, cities.name
FROM customers
JOIN cities ON cities.id = customers.city_id
WHERE customers.id <= 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name |  last_name   |    name
------------+--------------+-------------
 Μαρία      | Παπαδοπούλου | Αθήνα
 Γιώργος    | Παπαδόπουλος | Θεσσαλονίκη
 Ελένη      | Νικολάου     | Αθήνα
 Κώστας     | Δημητρίου    | Πάτρα
 Αναστασία  | Γεωργίου     | Ηράκλειο
(5 rows)
```

Η PostgreSQL εξετάζει κάθε ζεύγος γραμμών από τους δύο πίνακες και κρατά όσα κάνουν τη συνθήκη του `ON` αληθή. Το `JOIN` χωρίς άλλη λέξη είναι `INNER JOIN`. Οι δύο γραφές είναι ισοδύναμες.



Σχήμα 7.1 · Το INNER JOIN κρατά μόνο τα ζεύγη όπου η συνθήκη είναι αληθής. Ο πελάτης χωρίς πόλη και η πόλη χωρίς πελάτες δεν εμφανίζονται.

Το σχήμα δείχνει τι μένει έξω. Ο πελάτης 10 έχει `city_id` `NULL`. Η σύγκριση `NULL = 1` είναι άγνωστη, όπως είδες στο κεφάλαιο 3, οπότε ο πελάτης δεν ταιριάζει με καμία πόλη. Η Κοζάνη δεν έχει πελάτες, οπότε κανένα ζεύγος δεν την περιέχει. Το `INNER JOIN` χάνει και τους δύο. Το επόμενο κεφάλαιο δείχνει πώς τους κρατάς.

Aliases πινάκων

Το `customers.first_name` είναι σαφές αλλά μακρύ. Δίνεις σε κάθε πίνακα ένα σύντομο alias στο `FROM` και το χρησιμοποιείς παντού αλλού.

SQL

```
SELECT c.first_name, c.last_name, t.name AS city
FROM customers AS c
JOIN cities AS t ON t.id = c.city_id
WHERE c.id <= 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name |  last_name   |    city
------------+--------------+-------------
 Μαρία      | Παπαδοπούλου | Αθήνα
 Γιώργος    | Παπαδόπουλος | Θεσσαλονίκη
 Ελένη      | Νικολάου     | Αθήνα
 Κώστας     | Δημητρίου    | Πάτρα
 Αναστασία  | Γεωργίου     | Ηράκλειο
(5 rows)
```

Όταν μια στήλη υπάρχει και στους δύο πίνακες, το alias είναι υποχρεωτικό. Και ο `customers` και ο `cities` έχουν στήλη `id`.

SQL

```
SELECT id, first_name, name
FROM customers AS c
JOIN cities AS t ON t.id = c.city_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  column reference "id" is ambiguous
LINE 1: SELECT id, first_name, name
               ^
```

Γράφε το alias σε κάθε στήλη ενός query με join, ακόμη κι όταν δεν είναι υποχρεωτικό. Όποιος το διαβάσει ξέρει αμέσως από ποιον πίνακα έρχεται κάθε στήλη. Επιπλέον το query δεν σπάει αν αύριο κάποιος προσθέσει στήλη με το ίδιο όνομα στον άλλο πίνακα.

Ένα προς πολλά

Ένας πελάτης έχει πολλές παραγγελίες. Όταν ενώνεις τον `orders` με τον `customers`, κάθε παραγγελία βρίσκει έναν πελάτη και το αποτέλεσμα έχει μία γραμμή ανά παραγγελία. Τα στοιχεία του πελάτη επαναλαμβάνονται.

SQL

```
SELECT o.id AS order_id, o.created_at::date AS ordered_on, c.first_name, c.last_name
FROM orders AS o
JOIN customers AS c ON c.id = o.customer_id
WHERE c.id = 13
ORDER BY o.created_at;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | ordered_on | first_name | last_name
----------+------------+------------+-----------
       20 | 2025-05-04 | Κατερίνα   | Σταύρου
       35 | 2025-09-03 | Κατερίνα   | Σταύρου
       40 | 2025-09-19 | Κατερίνα   | Σταύρου
       41 | 2025-09-21 | Κατερίνα   | Σταύρου
       82 | 2026-02-04 | Κατερίνα   | Σταύρου
(5 rows)
```

Η κατεύθυνση μετράει όταν σκέφτεσαι πόσες γραμμές θα βγουν. Από την πλευρά του «ένα» κάθε γραμμή μπορεί να πολλαπλασιαστεί. Από την πλευρά του «πολλά» κάθε γραμμή βρίσκει το πολύ ένα ταίρι. Αυτό θα γίνει κρίσιμο όταν αρχίσουμε να αθροίζουμε στο κεφάλαιο 10.

Πολλά joins στη σειρά

Κάθε `JOIN` προσθέτει έναν πίνακα στο αποτέλεσμα των προηγούμενων. Οι γραμμές μιας παραγγελίας γίνονται ευανάγνωστες όταν φέρεις το όνομα του προϊόντος και της κατηγορίας του.

SQL

```
SELECT oi.order_id, p.name AS product, cat.name AS category,
       oi.quantity, oi.unit_price
FROM order_items AS oi
JOIN products AS p ON p.id = oi.product_id
JOIN categories AS cat ON cat.id = p.category_id
WHERE oi.order_id IN (137, 151)
ORDER BY oi.order_id, p.name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 order_id |              product                |     category     | quantity | unit_price
----------+-------------------------------------+------------------+----------+------------
      137 | Ηχείο Bluetooth                     | Ήχος             |        1 |      79.00
      137 | Καθαρός κώδικας στην πράξη          | Προγραμματισμός  |        1 |      38.00
      137 | Μαθαίνω Python                      | Προγραμματισμός  |        1 |      32.00
      137 | Σχεδίαση βάσεων δεδομένων           | Βάσεις δεδομένων |        1 |      36.00
      151 | Βίος και πολιτεία του Αλέξη Ζορμπά  | Λογοτεχνία       |        1 |      16.90
      151 | PostgreSQL σε βάθος                 | Βάσεις δεδομένων |        1 |      45.00
      151 | Webcam HD                           | Περιφερειακά     |        1 |      59.00
(7 rows)
```

Ο πίνακας `order_items` έχει και `unit_price` και `products.price`. Δεν είναι διπλή πληροφορία. Η `unit_price` είναι η τιμή τη στιγμή της αγοράς. Αν το προϊόν ακριβύνει αύριο, οι παλιές παραγγελίες δεν πρέπει να αλλάξουν.

Η σειρά των `INNER JOIN` δεν αλλάζει το αποτέλεσμα. Ο planner διαλέγει μόνος του με ποια σειρά θα τους εκτελέσει. Γράψε τους με τη σειρά που βοηθά τον αναγνώστη, συνήθως ακολουθώντας τα foreign keys.

Ο ίδιος πίνακας δύο φορές

Μια παραγγελία έχει πόλη αποστολής και ο πελάτης έχει πόλη κατοικίας. Και οι δύο δείχνουν στον `cities`. Για να πάρεις και τα δύο ονόματα ενώνεις τον `cities` δύο φορές, με διαφορετικό alias κάθε φορά.

SQL

```sql
SELECT o.id, home.name AS home_city, ship.name AS ship_city
FROM orders AS o
JOIN customers AS c ON c.id = o.customer_id
JOIN cities AS home ON home.id = c.city_id
JOIN cities AS ship ON ship.id = o.shipping_city_id
WHERE home.id <> ship.id
ORDER BY o.id
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |  home_city  | ship_city
----+-------------+-----------
  7 | Θεσσαλονίκη | Βόλος
 21 | Αθήνα       | Ιωάννινα
 31 | Θεσσαλονίκη | Χανιά
 38 | Θεσσαλονίκη | Βόλος
 41 | Χανιά       | Καλαμάτα
(5 rows)
```

Κάθε alias είναι ξεχωριστό αντίγραφο του πίνακα μέσα στο query. Αυτή η ιδέα, ο ίδιος πίνακας με δύο ρόλους, είναι η βάση του self join στο κεφάλαιο 9.

USING και NATURAL JOIN

Όταν οι στήλες σύνδεσης έχουν το ίδιο όνομα και στους δύο πίνακες, το `USING` είναι συντομογραφία. Στο σχήμα μας αυτό ισχύει για το `order_id` ανάμεσα στον `order_items` και στον `payments`.

SQL

```
SELECT order_id, oi.product_id, pay.amount
FROM order_items AS oi
JOIN payments AS pay USING (order_id)
WHERE order_id = 7;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | product_id | amount
----------+------------+--------
        7 |         25 |  50.00
        7 |         25 | 524.00
        7 |         22 |  50.00
        7 |         22 | 524.00
        7 |          9 |  50.00
        7 |          9 | 524.00
(6 rows)
```

Με `USING` η στήλη `order_id` εμφανίζεται μία φορά στο αποτέλεσμα και γράφεται χωρίς alias. Το αποτέλεσμα όμως έχει έξι γραμμές, ενώ η παραγγελία έχει τρία προϊόντα και δύο πληρωμές. Κάθε γραμμή προϊόντος ενώθηκε με κάθε πληρωμή. Κράτα αυτή την εικόνα. Είναι το πρόβλημα του κεφαλαίου 11.

ΠΑΓΙΔΑ

Το `NATURAL JOIN` ενώνει αυτόματα σε όλες τις στήλες με κοινό όνομα. Στον `customers` και στον `orders` κοινές είναι η `id` και η `created_at`, οπότε το `NATURAL JOIN` θα απαιτούσε την ίδια τιμή και στις δύο και θα έβγαζε ανοησίες. Μην το χρησιμοποιείς.

ΚΡΑΤΑ ΑΥΤΟ

Το `JOIN ... ON` κρατά τα ζεύγη γραμμών που κάνουν τη συνθήκη αληθή. Γραμμές χωρίς ταίρι χάνονται. Δώσε alias σε κάθε πίνακα και χρησιμοποίησέ το σε κάθε στήλη. Από την πλευρά του «ένα» μιας σχέσης ένα προς πολλά, οι γραμμές πολλαπλασιάζονται.

Ασκήσεις

ΑΣΚΗΣΗ 7.1 Πελάτες του 2026 και πόλεις

Εμφάνισε όνομα, επώνυμο και όνομα πόλης των πελατών που εγγράφηκαν το 2026, ταξινομημένους κατά ημερομηνία εγγραφής.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 first_name |  last_name  |    city
------------+-------------+-------------
 Πέτρος     | Σαββίδης    | Θεσσαλονίκη
 Ηλίας      | Τζαννετάκης | Ηράκλειο
 Όλγα       | Βλάχου      | Αθήνα
(3 rows)
```

ΑΣΚΗΣΗ 7.2 Περιφερειακά

Εμφάνισε όνομα και τιμή των προϊόντων της κατηγορίας με όνομα `Περιφερειακά`, χωρίς να γράψεις τον αριθμό της κατηγορίας.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
         name           | price
------------------------+--------
 Ασύρματο ποντίκι       |  24.90
 Webcam HD              |  59.00
 Μηχανικό πληκτρολόγιο  |  89.00
 USB-C docking station  | 139.00
 Οθόνη 27" 4K           | 329.00
(5 rows)
```

ΑΣΚΗΣΗ 7.3 Λογαριασμός παραγγελίας

Για την παραγγελία 144 εμφάνισε το όνομα κάθε προϊόντος, την ποσότητα, την τιμή μονάδας και το σύνολο της γραμμής με όνομα `line_total`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       name       | quantity | unit_price | line_total
------------------+----------+------------+------------
 Ασύρματο ποντίκι |        1 |      24.90 |      24.90
 Ματωμένα χώματα  |        2 |      15.20 |      30.40
 Σετ μαχαιριών    |        2 |      64.00 |     128.00
(3 rows)
```

ΑΣΚΗΣΗ 7.4 Παραγγελίες από την Κρήτη

Εμφάνισε `id` παραγγελίας, όνομα πελάτη και πόλη κατοικίας για τις παραγγελίες του Μαΐου 2026 από πελάτες που μένουν στην περιφέρεια Κρήτης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | first_name |  last_name   |   city
-----+------------+--------------+----------
 115 | Ηλίας      | Τζαννετάκης  | Ηράκλειο
 123 | Ηλίας      | Τζαννετάκης  | Ηράκλειο
 127 | Λευτέρης   | Φραγκιαδάκης | Χανιά
 131 | Ηλίας      | Τζαννετάκης  | Ηράκλειο
(4 rows)
```

ΑΣΚΗΣΗ 7.5 Προμηθευτές ακουστικών

Εμφάνισε το όνομα κάθε προμηθευτή που φέρνει το προϊόν `Ακουστικά ANC` και την τιμή προμήθειας.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     supplier     | supply_price
------------------+--------------
 Ήχος & Εικόνα ΑΕ |       120.00
 Techno Import    |       124.80
(2 rows)
```

ΑΣΚΗΣΗ 7.6 Πληρωμές επιστροφών

Εμφάνισε `id` παραγγελίας, ποσό, μέθοδο και ημερομηνία πληρωμής για όλες τις πληρωμές των παραγγελιών με κατάσταση `refunded`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  |  amount  | method |  paid_on
-----+----------+--------+------------
  55 |  1190.00 | card   | 2025-11-10
  55 | -1190.00 | card   | 2025-11-26
  97 |    90.00 | paypal | 2026-03-13
  97 |   -90.00 | paypal | 2026-03-26
 106 |   339.00 | paypal | 2026-04-04
 106 |  -339.00 | paypal | 2026-04-17
(6 rows)
```

ΑΣΚΗΣΗ 7.7 Κριτικές με χαμηλή βαθμολογία

Εμφάνισε βαθμολογία, όνομα προϊόντος, όνομα πελάτη και τίτλο για τις κριτικές με βαθμολογία 1 ή 2.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 rating |      product       | first_name |    title
--------+--------------------+------------+-------------
      1 | Ασύρματο ποντίκι   | Σοφία      | Χάλασε
      2 | Καφετιέρα espresso | Παναγιώτης | Διαρροή
      2 | Laptop Aero 14     | Σοφία      | Ζεσταίνεται
      2 | Laptop Student 15  | Κατερίνα   | Αργό
(4 rows)
```

ΑΣΚΗΣΗ 7.8 Αποστολή εκτός περιφέρειας

Βρες τις παραγγελίες που στάλθηκαν σε πόλη άλλης περιφέρειας από την περιφέρεια κατοικίας του πελάτη. Εμφάνισε `id` παραγγελίας και τις δύο περιφέρειες.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  |    home_region     |  ship_region
-----+--------------------+---------------
   7 | Κεντρική Μακεδονία | Θεσσαλία
  21 | Αττική             | Ήπειρος
  31 | Κεντρική Μακεδονία | Κρήτη
  38 | Κεντρική Μακεδονία | Θεσσαλία
  41 | Κρήτη              | Πελοπόννησος
  53 | Ήπειρος            | Αττική
  78 | Κεντρική Μακεδονία | Δυτική Ελλάδα
  83 | Κρήτη              | Πελοπόννησος
  87 | Κρήτη              | Αττική
  89 | Κεντρική Μακεδονία | Κρήτη
  95 | Κρήτη              | Δυτική Ελλάδα
  99 | Δυτική Ελλάδα      | Κρήτη
 129 | Αττική             | Ήπειρος
 133 | Δυτική Ελλάδα      | Πελοπόννησος
 …
(17 rows)
```

ΚΕΦΑΛΑΙΟ 8

OUTER JOIN και anti join

Το `LEFT JOIN` κρατά κάθε γραμμή του αριστερού πίνακα, ακόμη κι αν δεν βρει ταίρι. Όπου δεν βρει, οι στήλες του δεξιού πίνακα γεμίζουν με `NULL`. Με αυτό απαντάς και την αντίθετη ερώτηση, ποιες γραμμές δεν έχουν ταίρι.

LEFT JOIN

Στο προηγούμενο κεφάλαιο ο πελάτης 10 χάθηκε από τη λίστα πελατών με πόλεις, επειδή δεν έχει πόλη. Το `LEFT JOIN` τον κρατά.

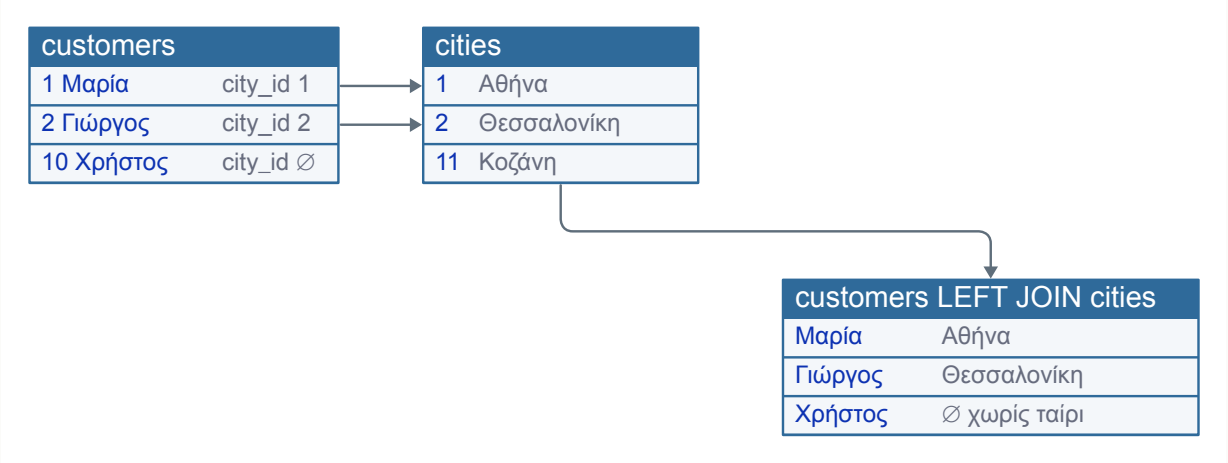
SQL

```
SELECT c.id, c.first_name, t.name AS city
FROM customers AS c
LEFT JOIN cities AS t ON t.id = c.city_id
WHERE c.id BETWEEN 8 AND 12
ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |   city
----+------------+----------
  8 | Παναγιώτης | Αθήνα
  9 | Σοφία      | Βόλος
 10 | Χρήστος    | ø
 11 | Ιωάννα     | Ιωάννινα
 12 | Αλέξανδρος | Αθήνα
(5 rows)
```

Ο κανόνας είναι απλός. Κάθε γραμμή του αριστερού πίνακα, αυτού που γράφεται πριν από το `LEFT JOIN`, εμφανίζεται τουλάχιστον μία φορά. Αν βρει ταίρια, εμφανίζεται μία φορά για καθένα, όπως στο `INNER JOIN`. Αν δεν βρει κανένα, εμφανίζεται μία φορά με `NULL` σε όλες τις στήλες του δεξιού πίνακα.



Σχήμα 8.1 · Στο LEFT JOIN ο πελάτης χωρίς πόλη μένει στο αποτέλεσμα με ∅ στις στήλες της πόλης. Η πόλη χωρίς πελάτες εξακολουθεί να λείπει.

Οι λέξεις `OUTER` και `INNER` είναι προαιρετικές. Το `LEFT JOIN` και το `LEFT OUTER JOIN` είναι το ίδιο.

Οι παραγγελίες με ή χωρίς πληρωμή

Το `LEFT JOIN` είναι το σωστό εργαλείο όταν η σχέση είναι προαιρετική. Μια παραγγελία μπορεί να έχει καμία, μία ή δύο πληρωμές.

SQL

```
SELECT o.id, o.status, pay.amount, pay.method
FROM orders AS o
LEFT JOIN payments AS pay ON pay.order_id = o.id
WHERE o.customer_id = 6
ORDER BY o.id, pay.amount;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |  status   |  amount  |  method
-----+-----------+----------+----------
   2 | delivered |    38.00 | paypal
  25 | cancelled |        ∅ | ∅
  26 | delivered |   219.00 | card
  39 | delivered |    32.00 | paypal
  42 | delivered |   459.00 | paypal
  45 | cancelled |        ∅ | ∅
  55 | refunded  | -1190.00 | card
  55 | refunded  |  1190.00 | card
 102 | delivered |    20.50 | bank
 102 | delivered |    50.00 | giftcard
 121 | cancelled |        ∅ | ∅
 143 | shipped   |   235.00 | card
(12 rows)
```

Οι ακυρωμένες παραγγελίες εμφανίζονται μία φορά με ∅. Η επιστροφή 55 και η παραγγελία 102 με δωροκάρτα εμφανίζονται δύο φορές, μία για κάθε πληρωμή. Το `LEFT JOIN` εξακολουθεί να πολλαπλασιάζει γραμμές. Απλώς δεν αφήνει καμία να χαθεί.

Anti join. Όσα δεν έχουν ταίρι

Η ερώτηση «ποιοι πελάτες δεν έχουν κάνει ποτέ παραγγελία» απαντιέται με `LEFT JOIN` και έναν έλεγχο. Οι πελάτες χωρίς παραγγελίες είναι αυτοί όπου οι στήλες του `orders` βγήκαν `NULL`.

SQL

```
SELECT c.id, c.first_name, c.last_name
FROM customers AS c
LEFT JOIN orders AS o ON o.customer_id = c.id
WHERE o.id IS NULL
ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | last_name
----+------------+------------
 10 | Χρήστος    | Μιχαηλίδης
 26 | Ραλλού     | Κοντού
 30 | Όλγα       | Βλάχου
(3 rows)
```

Ο έλεγχος γίνεται σε στήλη που δεν μπορεί να είναι `NULL` σε μια πραγματική γραμμή, εδώ το primary key `o.id`. Αν έλεγχες μια στήλη όπως η `o.coupon_code`, θα έπαιρνες και τους πελάτες που έχουν παραγγελίες χωρίς κουπόνι. Αυτό το μοτίβο λέγεται **anti join**. Θα δεις έναν δεύτερο τρόπο να το γράψεις, με `NOT EXISTS`, στο κεφάλαιο 13.

Με τον ίδιο τρόπο βρίσκεις τα προϊόντα που δεν έχουν πουληθεί ποτέ.

SQL

```
SELECT p.id, p.name
FROM products AS p
LEFT JOIN order_items AS oi ON oi.product_id = p.id
WHERE oi.order_id IS NULL;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | name
----+-------
 30 | Πικάπ
(1 row)
```

ON ή WHERE

Στο `INNER JOIN` δεν έχει σημασία αν μια συνθήκη μπει στο `ON` ή στο `WHERE`. Στο `LEFT JOIN` έχει τεράστια. Θέλεις κάθε πελάτη της Θεσσαλονίκης μαζί με τις ακυρωμένες παραγγελίες του, αν έχει.

SQL

```
SELECT c.id, c.first_name, o.id AS cancelled_order
FROM customers AS c
LEFT JOIN orders AS o ON o.customer_id = c.id
WHERE c.city_id = 2
  AND o.status = 'cancelled';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | cancelled_order
----+------------+-----------------
  2 | Γιώργος    |               4
  6 | Νίκος      |              25
  6 | Νίκος      |              45
  6 | Νίκος      |             121
(4 rows)
```

Βγήκαν μόνο οι πελάτες που έχουν ακυρωμένη παραγγελία. Το `WHERE` εκτελείται μετά το join. Για τους υπόλοιπους πελάτες το `o.status` είναι είτε κάτι άλλο είτε `NULL`, οπότε το `WHERE` πέταξε αυτές τις γραμμές. Το `LEFT JOIN` έγινε στην πράξη `INNER JOIN`. Η συνθήκη πάνω στον δεξιό πίνακα ανήκει στο `ON`.

SQL

```
SELECT c.id, c.first_name, o.id AS cancelled_order
FROM customers AS c
LEFT JOIN orders AS o ON o.customer_id = c.id AND o.status = 'cancelled'
WHERE c.city_id = 2
ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | cancelled_order
----+------------+-----------------
  2 | Γιώργος    |               4
  6 | Νίκος      |              25
  6 | Νίκος      |              45
  6 | Νίκος      |             121
 14 | Βασίλης    |               ∅
 21 | Giorgos    |               ∅
 24 | Ζωή        |               ∅
 27 | Πέτρος     |               ∅
(8 rows)
```

Τώρα το `ON` λέει «ταίριαξε με τις ακυρωμένες παραγγελίες του πελάτη». Όποιος δεν έχει καμία μένει με ∅. Το `WHERE c.city_id = 2` αφορά τον αριστερό πίνακα και σωστά μένει στο `WHERE`.

ΠΑΓΙΔΑ

Σε `LEFT JOIN`, οι συνθήκες για τον δεξιό πίνακα μπαίνουν στο `ON`. Μια συνθήκη στο `WHERE` πάνω σε στήλη του δεξιού πίνακα πετά τις γραμμές χωρίς ταίρι. Εξαίρεση είναι ο έλεγχος `IS NULL` του anti join, που γράφεται στο `WHERE` ακριβώς επειδή θέλει τις γραμμές χωρίς ταίρι.

Αλυσίδες με LEFT JOIN

Όταν ένας πίνακας έρχεται από `LEFT JOIN`, κάθε επόμενο join πάνω του πρέπει συνήθως να είναι κι αυτό `LEFT JOIN`. Θέλεις κάθε παραγγελία του πελάτη 12 από τον Μάιο του 2026 μαζί με τα προϊόντα της.

SQL

```
SELECT o.id, o.status, p.name
FROM orders AS o
LEFT JOIN order_items AS oi ON oi.order_id = o.id
JOIN products AS p ON p.id = oi.product_id
WHERE o.customer_id = 12 AND o.created_at >= '2026-05-01'
ORDER BY o.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |  status   |           name
-----+-----------+--------------------------
 116 | delivered | Laptop Student 15
 134 | delivered | Φωτιστικό γραφείου LED
 149 | shipped   | Σχεδίαση βάσεων δεδομένων
(3 rows)
```

Η παραγγελία 162 είναι pending χωρίς γραμμές ακόμη. Το `LEFT JOIN` την κράτησε με `oi.product_id` `NULL`. Το επόμενο `INNER JOIN` όμως απαιτεί `p.id = NULL`, που δεν είναι ποτέ αληθές. Έτσι η παραγγελία χάθηκε. Η λύση είναι να μείνει `LEFT` και το δεύτερο join.

SQL

```
SELECT o.id, o.status, p.name
FROM orders AS o
LEFT JOIN order_items AS oi ON oi.order_id = o.id
LEFT JOIN products AS p ON p.id = oi.product_id
WHERE o.customer_id = 12 AND o.created_at >= '2026-05-01'
ORDER BY o.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |  status   |           name
-----+-----------+--------------------------
 116 | delivered | Laptop Student 15
 134 | delivered | Φωτιστικό γραφείου LED
 149 | shipped   | Σχεδίαση βάσεων δεδομένων
 162 | pending   | ∅
(4 rows)
```

RIGHT JOIN και FULL JOIN

Το `RIGHT JOIN` είναι το καθρέφτισμα του `LEFT JOIN`. Κρατά κάθε γραμμή του δεξιού πίνακα. Το `a RIGHT JOIN b` είναι ίδιο με το `b LEFT JOIN a`. Στην πράξη οι περισσότεροι γράφουν πάντα `LEFT JOIN` και βάζουν πρώτο τον πίνακα που θέλουν να κρατήσουν ολόκληρο.

Το `FULL JOIN` κρατά τις γραμμές χωρίς ταίρι και από τις δύο πλευρές. Είναι χρήσιμο για συμφωνία δύο λιστών. Εδώ βρίσκει και τις κατηγορίες χωρίς προϊόντα και τα προϊόντα χωρίς κατηγορία.

SQL

```
SELECT cat.name AS category, p.name AS product
FROM categories AS cat
FULL JOIN products AS p ON p.category_id = cat.id
WHERE cat.id IS NULL OR p.id IS NULL
ORDER BY cat.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  category   |    product
-------------+---------------
 Βιβλία      | ∅
 Ηλεκτρονικά | ∅
 Υπολογιστές | ∅
 Σπίτι       | ∅
 ∅           | Δωροκάρτα 50€
(5 rows)
```

Οι τέσσερις κατηγορίες είναι οι γονικές του δέντρου. Τα προϊόντα ανήκουν στις υποκατηγορίες τους. Η δωροκάρτα δεν ανήκει πουθενά.

ΚΡΑΤΑ ΑΥΤΟ

Το `LEFT JOIN` κρατά κάθε γραμμή του αριστερού πίνακα και γεμίζει με `NULL` όπου δεν βρίσκει ταίρι. Το anti join είναι `LEFT JOIN` και `WHERE δεξί.id IS NULL`. Οι συνθήκες για τον δεξιό πίνακα μπαίνουν στο `ON`. Μετά από ένα `LEFT JOIN` γίνονται `LEFT` και τα επόμενα joins της ίδιας αλυσίδας.

Ασκήσεις

ΑΣΚΗΣΗ 8.1 Πόλη ή άγνωστη

Εμφάνισε `id`, όνομα και πόλη όλων των πελατών που εγγράφηκαν το 2026. Όπου η πόλη λείπει γράψε `άγνωστη`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |    city
----+------------+-------------
 27 | Πέτρος     | Θεσσαλονίκη
 28 | Δέσποινα   | άγνωστη
 29 | Ηλίας      | Ηράκλειο
 30 | Όλγα       | Αθήνα
(4 rows)
```

ΑΣΚΗΣΗ 8.2 Πόλεις χωρίς πελάτες

Βρες τις πόλεις στις οποίες δεν μένει κανένας πελάτης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |  name
----+--------
 11 | Κοζάνη
(1 row)
```

ΑΣΚΗΣΗ 8.3 Παραγγελίες χωρίς γραμμές

Βρες τις παραγγελίες που δεν έχουν καμία γραμμή στον `order_items`. Εμφάνισε `id`, `status` και `customer_id`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | status  | customer_id
-----+---------+-------------
 162 | pending |          12
(1 row)
```

ΑΣΚΗΣΗ 8.4 Παραγγελίες χωρίς πληρωμή

Βρες τις παραγγελίες που δεν έχουν πληρωμή και δεν είναι ακυρωμένες. Εμφάνισε `id`, `status` και `created_at`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | status  |       created_at
-----+---------+------------------------
 150 | shipped | 2026-06-15 18:06:00+03
 157 | pending | 2026-06-23 09:56:00+03
 159 | pending | 2026-06-24 19:00:00+03
 162 | pending | 2026-06-28 17:18:00+03
(4 rows)
```

ΑΣΚΗΣΗ 8.5 Πελάτες και κουπόνια

Εμφάνισε κάθε πελάτη της Αθήνας (πόλη 1) με όνομα και τα `id` των παραγγελιών του που χρησιμοποίησαν κουπόνι. Οι πελάτες χωρίς τέτοια παραγγελία πρέπει να εμφανίζονται μία φορά με ∅. Ταξινόμησε κατά πελάτη και παραγγελία.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | order_id | coupon_code
----+------------+----------+-------------
  1 | Μαρία      |       11 | WELCOME10
  1 | Μαρία      |       17 | WELCOME10
  1 | Μαρία      |       58 | BLACKFRIDAY
  3 | Ελένη      |        ∅ | ∅
  8 | Παναγιώτης |       13 | WELCOME10
  8 | Παναγιώτης |      148 | WELCOME10
 12 | Αλέξανδρος |       30 | WELCOME10
 19 | Αικατερίνη |       66 | WELCOME10
 23 | Στέλιος    |      108 | WELCOME10
 30 | Όλγα       |        ∅ | ∅
(10 rows)
```

ΑΣΚΗΣΗ 8.6 Υπάλληλοι που είναι και πελάτες

Ο πίνακας `employees` έχει στήλη `email`. Εμφάνισε κάθε υπάλληλο του τμήματος Πωλήσεις και, όπου υπάρχει, τον πελάτη με το ίδιο email. Δεν πρέπει να χαθεί κανένας υπάλληλος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     full_name      |           email            | customer_id
--------------------+----------------------------+-------------
 Σοφία Κωνσταντίνου | sofia.k@shop.example.gr    |           ∅
 Ιωάννα Ρήγα        | ioanna.r@shop.example.gr   |           ∅
 Στέλιος Παππάς     | stelios.p@shop.example.gr  |           ∅
 Βασιλική Ντόκου    | ∅                          |           ∅
(4 rows)
```

ΑΣΚΗΣΗ 8.7 Κατηγορίες και προϊόντα

Εμφάνισε όλες τις κατηγορίες των ηλεκτρονικών (τα `id` από 6 έως 10) με το όνομα κάθε ενεργού προϊόντος τους. Οι κατηγορίες χωρίς ενεργό προϊόν πρέπει να εμφανίζονται με ∅. Ταξινόμησε κατά `id` κατηγορίας και όνομα προϊόντος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |   category    |        product
----+---------------+-----------------------
  6 | Ηλεκτρονικά   | ∅
  7 | Υπολογιστές   | ∅
  8 | Laptops       | Laptop Aero 14
  8 | Laptops       | Laptop Pro 16
  8 | Laptops       | Laptop Student 15
  9 | Περιφερειακά  | Ασύρματο ποντίκι
  9 | Περιφερειακά  | Μηχανικό πληκτρολόγιο
  9 | Περιφερειακά  | Οθόνη 27" 4K
  9 | Περιφερειακά  | USB-C docking station
  9 | Περιφερειακά  | Webcam HD
 10 | Ήχος          | Ακουστικά ANC
 10 | Ήχος          | Ηχείο Bluetooth
 10 | Ήχος          | Μικρόφωνο USB
 10 | Ήχος          | Πικάπ
(14 rows)
```

ΑΣΚΗΣΗ 8.8 Πελάτες χωρίς κριτική

Βρες τους πελάτες που έχουν τουλάχιστον μία παραγγελία `delivered` αλλά δεν έχουν γράψει καμία κριτική. Εμφάνισε `id` και όνομα χωρίς επαναλήψεις.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name
----+------------
 11 | Ιωάννα
 14 | Βασίλης
 15 | Ευαγγελία
 17 | Φωτεινή
 18 | Σπύρος
 19 | Αικατερίνη
 20 | Μιχάλης
 21 | Giorgos
 22 | Άννα
 23 | Στέλιος
 24 | Ζωή
 25 | Λευτέρης
 27 | Πέτρος
 28 | Δέσποινα
 …
(15 rows)
```

ΚΕΦΑΛΑΙΟ 9

Self join, CROSS JOIN και joins με εύρος

Ένα join δεν χρειάζεται δύο διαφορετικούς πίνακες ούτε συνθήκη ισότητας. Ο ίδιος πίνακας μπορεί να ενωθεί με τον εαυτό του και η συνθήκη μπορεί να είναι οποιαδήποτε έκφραση. Χωρίς συνθήκη, το join δίνει όλους τους συνδυασμούς.

Self join

Στον `employees` κάθε υπάλληλος έχει `manager_id`, που δείχνει σε άλλη γραμμή του ίδιου πίνακα. Για να δεις δίπλα σε κάθε υπάλληλο το όνομα του προϊσταμένου του ενώνεις τον πίνακα με τον εαυτό του. Χρειάζονται δύο aliases, ένα για κάθε ρόλο, όπως στο κεφάλαιο 7 με τις δύο πόλεις.

SQL

```
SELECT e.full_name AS employee, e.title, m.full_name AS manager
FROM employees AS e
JOIN employees AS m ON m.id = e.manager_id
WHERE e.department = 'Τεχνολογία'
ORDER BY e.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
      employee      |       title        |      manager
--------------------+--------------------+-------------------
 Νίκος Αλεξίου      | CTO                | Ελένη Μαυρίδου
 Κατερίνα Ζαφειρίου | Backend Developer  | Νίκος Αλεξίου
 Άγγελος Φωτίου     | Frontend Developer | Νίκος Αλεξίου
 Μαρίνα Σταθοπούλου | Data Engineer      | Νίκος Αλεξίου
 Πέτρος Χατζής      | Junior Developer   | Κατερίνα Ζαφειρίου
(5 rows)
```

Κάθε γραμμή του `e` βρήκε τη γραμμή του προϊσταμένου της στο `m`. Το `WHERE` κοιτάζει το τμήμα του `e`, οπότε ο CTO εμφανίζεται με προϊστάμενο από τη Διοίκηση. Το `INNER JOIN` όμως χάνει όσους δεν έχουν `manager_id`. Για να τους δεις όλους χρειάζεσαι `LEFT JOIN`.

SQL

```
SELECT e.id, e.full_name AS employee, m.full_name AS manager
FROM employees AS e
LEFT JOIN employees AS m ON m.id = e.manager_id
ORDER BY e.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |       employee        |        manager
----+-----------------------+-----------------------
  1 | Ελένη Μαυρίδου        | ∅
  2 | Νίκος Αλεξίου         | Ελένη Μαυρίδου
  3 | Σοφία Κωνσταντίνου    | Ελένη Μαυρίδου
  4 | Δημήτρης Λαμπρόπουλος | Ελένη Μαυρίδου
  5 | Κατερίνα Ζαφειρίου    | Νίκος Αλεξίου
  6 | Άγγελος Φωτίου        | Νίκος Αλεξίου
  7 | Μαρίνα Σταθοπούλου    | Νίκος Αλεξίου
  8 | Πέτρος Χατζής         | Κατερίνα Ζαφειρίου
  9 | Ιωάννα Ρήγα           | Σοφία Κωνσταντίνου
 10 | Στέλιος Παππάς        | Σοφία Κωνσταντίνου
 11 | Βασιλική Ντόκου       | Ιωάννα Ρήγα
 12 | Γιάννης Κορρές        | Δημήτρης Λαμπρόπουλος
 13 | Θανάσης Μπέης         | Δημήτρης Λαμπρόπουλος
 14 | Χριστίνα Αθανασίου    | Γιάννης Κορρές
 …
(15 rows)
```

Οι υπάλληλοι 1 και 15 δεν έχουν προϊστάμενο και εμφανίζονται με ∅.

Κάθε επόμενο επίπεδο της ιεραρχίας θέλει ένα ακόμη join. Ο προϊστάμενος του προϊσταμένου είναι ένα τρίτο αντίγραφο του πίνακα.

SQL

```
SELECT e.full_name AS employee, m.full_name AS manager, mm.full_name AS managers_manager
FROM employees AS e
JOIN employees AS m ON m.id = e.manager_id
JOIN employees AS mm ON mm.id = m.manager_id
WHERE e.department = 'Αποθήκη';
```

ΑΠΟΤΕΛΕΣΜΑ

```
      employee      |        manager         |   managers_manager
--------------------+------------------------+-----------------------
 Γιάννης Κορρές     | Δημήτρης Λαμπρόπουλος | Ελένη Μαυρίδου
 Θανάσης Μπέης      | Δημήτρης Λαμπρόπουλος | Ελένη Μαυρίδου
 Χριστίνα Αθανασίου | Γιάννης Κορρές        | Δημήτρης Λαμπρόπουλος
(3 rows)
```

Αν η ιεραρχία έχει άγνωστο βάθος, δεν ξέρεις πόσα joins να γράψεις. Εκεί χρειάζεται το recursive CTE του κεφαλαίου 22.

Σύγκριση γραμμών του ίδιου πίνακα

Το self join χρησιμεύει και όταν θέλεις ζεύγη γραμμών που μοιάζουν. Οι πελάτες με ίδια ημερομηνία γέννησης και ίδιο επώνυμο σε ελληνικά ή λατινικά μπορεί να είναι διπλοεγγραφές. Ξεκινάμε από την ημερομηνία γέννησης.

SQL

```
SELECT a.id, a.first_name, a.last_name, b.id, b.first_name, b.last_name
FROM customers AS a
JOIN customers AS b ON b.birth_date = a.birth_date AND b.id <> a.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name   | id | first_name |  last_name
----+------------+--------------+----+------------+-------------
  2 | Γιώργος    | Παπαδόπουλος | 21 | Giorgos    | Papadopoulos
 21 | Giorgos    | Papadopoulos |  2 | Γιώργος    | Παπαδόπουλος
(2 rows)
```

Το ίδιο ζεύγος εμφανίστηκε δύο φορές, μία ως (2, 21) και μία ως (21, 2). Η συνθήκη `b.id <> a.id` αποκλείει μόνο το ζεύγος μιας γραμμής με τον εαυτό της. Για να πάρεις κάθε ζεύγος μία φορά απαιτείς σειρά.

SQL

```
SELECT a.id, a.first_name, a.last_name, b.id, b.first_name, b.last_name
FROM customers AS a
JOIN customers AS b ON b.birth_date = a.birth_date AND b.id > a.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name   | id | first_name |  last_name
----+------------+--------------+----+------------+-------------
  2 | Γιώργος    | Παπαδόπουλος | 21 | Giorgos    | Papadopoulos
(1 row)
```

Το κεφάλαιο 28 επιστρέφει σε αυτούς τους δύο πελάτες και τους εντοπίζει από το όνομα, με trigrams.

CROSS JOIN

Το `CROSS JOIN` δεν έχει συνθήκη. Ενώνει κάθε γραμμή του πρώτου πίνακα με κάθε γραμμή του δεύτερου. Με 3 και 4 γραμμές δίνει 12. Το αποτέλεσμα λέγεται **καρτεσιανό γινόμενο**.

Για να δείξουμε ένα μικρό παράδειγμα χρειαζόμαστε μια μικρή λίστα τιμών. Το `VALUES` φτιάχνει έναν προσωρινό πίνακα μέσα στο query. Κάθε παρένθεση είναι μια γραμμή και μετά το alias δίνεις ονόματα στις στήλες.

SQL

```
SELECT p.name, q.qty, p.price * q.qty AS total
FROM products AS p
CROSS JOIN (VALUES (1), (5), (10)) AS q(qty)
WHERE p.id IN (1, 2)
ORDER BY p.name, q.qty;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                | qty | total
------------------------------------+-----+--------
 Βίος και πολιτεία του Αλέξη Ζορμπά |   1 |  16.90
 Βίος και πολιτεία του Αλέξη Ζορμπά |   5 |  84.50
 Βίος και πολιτεία του Αλέξη Ζορμπά |  10 | 169.00
 Το Κιβώτιο                         |   1 |  14.50
 Το Κιβώτιο                         |   5 |  72.50
 Το Κιβώτιο                         |  10 | 145.00
(6 rows)
```

Ο τιμοκατάλογος για ποσότητες 1, 5 και 10 έχει έξι γραμμές, δύο προϊόντα επί τρεις ποσότητες. Το `CROSS JOIN` είναι χρήσιμο όταν θέλεις πραγματικά όλους τους συνδυασμούς. Στο κεφάλαιο 25 θα το χρησιμοποιήσουμε για να φτιάξουμε πλέγμα με κάθε ημέρα και κάθε κατηγορία, ώστε να φαίνονται και τα μηδενικά.

> **ΠΑΓΙΔΑ**
> Ένα `JOIN` που ξέχασε τη συνθήκη του ή που έχει λάθος συνθήκη συμπεριφέρεται σαν `CROSS JOIN`. Με δύο πίνακες 100.000 γραμμών το αποτέλεσμα έχει δέκα δισεκατομμύρια γραμμές. Όταν ένα query επιστρέφει πολύ περισσότερες γραμμές από όσες περίμενες, έλεγξε πρώτα τις συνθήκες των joins.

Η παλιά γραφή με κόμμα στο `FROM`, `FROM a, b WHERE a.x = b.y`, είναι `CROSS JOIN` που γίνεται join από τη συνθήκη του `WHERE`. Θα τη δεις σε παλιό κώδικα. Προτίμησε το `JOIN ... ON`, που κρατά τη συνθήκη δίπλα στον πίνακα.

Joins με εύρος

Η συνθήκη ενός join μπορεί να είναι οποιαδήποτε έκφραση που δίνει boolean. Ένα συνηθισμένο παράδειγμα είναι ένας πίνακας κλιμακίων. Η κατηγορία τιμής ενός προϊόντος δεν είναι ισότητα αλλά ανήκει σε ένα διάστημα.

SQL

```
SELECT p.name, p.price, b.label
FROM products AS p
JOIN (VALUES ('οικονομικό', 0, 50),
             ('μεσαίο', 50, 300),
             ('ακριβό', 300, 100000)) AS b(label, lo, hi)
  ON p.price >= b.lo AND p.price < b.hi
WHERE p.category_id = 9
ORDER BY p.price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name          | price  |   label
-----------------------+--------+------------
 Ασύρματο ποντίκι      |  24.90 | οικονομικό
 Webcam HD             |  59.00 | μεσαίο
 Μηχανικό πληκτρολόγιο |  89.00 | μεσαίο
 USB-C docking station | 139.00 | μεσαίο
 Οθόνη 27" 4K          | 329.00 | ακριβό
(5 rows)
```

Είναι το ίδιο αποτέλεσμα με το `CASE` του κεφαλαίου 5. Η διαφορά είναι ότι τα κλιμάκια είναι δεδομένα και όχι κώδικας. Αν ζουν σε έναν πραγματικό πίνακα, τα αλλάζεις χωρίς να αγγίξεις κανένα query.

Τα διαστήματα είναι μισάνοιχτα, όπως στο κεφάλαιο 6. Ένα προϊόν των 50 ευρώ ανήκει στο `μεσαίο` και μόνο εκεί. Αν τα διαστήματα επικαλύπτονταν, το προϊόν θα έβρισκε δύο ταίρια και θα εμφανιζόταν δύο φορές.

Ένα join με εύρος μπορεί να συνδέσει και δύο κανονικούς πίνακες. Εδώ βρίσκουμε για κάθε κριτική την παραγγελία του ίδιου πελάτη για το ίδιο προϊόν που είχε παραδοθεί πριν από την κριτική.

SQL

```
SELECT r.id AS review, r.created_at::date AS reviewed, o.id AS order_id,
       o.shipped_at::date AS shipped
FROM reviews AS r
JOIN orders AS o ON o.customer_id = r.customer_id
                AND o.shipped_at < r.created_at
JOIN order_items AS oi ON oi.order_id = o.id AND oi.product_id = r.product_id
WHERE r.product_id = 19
ORDER BY r.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 review |  reviewed  | order_id |  shipped
--------+------------+----------+------------
     12 | 2025-04-04 |       13 | 2025-03-30
     23 | 2025-06-28 |       26 | 2025-06-22
     24 | 2025-07-12 |       27 | 2025-06-30
     37 | 2025-12-07 |       58 | 2025-11-23
(4 rows)
```

ΚΡΑΤΑ ΑΥΤΟ

Self join σημαίνει ίδιος πίνακας με δύο aliases και δύο ρόλους. Για μοναδικά ζεύγη γράψε `b.id > a.id`. Το `CROSS JOIN` δίνει όλους τους συνδυασμούς. Η συνθήκη ενός join μπορεί να είναι διάστημα. Τότε τα διαστήματα δεν πρέπει να επικαλύπτονται.

Ασκήσεις

ΑΣΚΗΣΗ 9.1 Υπάλληλοι και προϊστάμενοι

Εμφάνισε για κάθε υπάλληλο του τμήματος Πωλήσεις το όνομά του, τη θέση του και το όνομα και τη θέση του προϊσταμένου του.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      employee      |      title      |      manager       |  manager_title
--------------------+-----------------+--------------------+-----------------
 Σοφία Κωνσταντίνου | Head of Sales   | Ελένη Μαυρίδου     | CEO
 Ιωάννα Ρήγα       | Account Manager | Σοφία Κωνσταντίνου | Head of Sales
 Στέλιος Παππάς    | Account Manager | Σοφία Κωνσταντίνου | Head of Sales
 Βασιλική Ντόκου   | Sales Intern    | Ιωάννα Ρήγα        | Account Manager
(4 rows)
```

ΑΣΚΗΣΗ 9.2 Διαφορά μισθού

Για κάθε υπάλληλο με προϊστάμενο και γνωστό μισθό, εμφάνισε το όνομά του, τον μισθό του, τον μισθό του προϊσταμένου και τη διαφορά τους με όνομα `gap`. Ταξινόμησε από τη μεγαλύτερη διαφορά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       full_name       | salary  | manager_salary |   gap
-----------------------+---------+----------------+---------
 Θανάσης Μπέης         | 1750.00 |        6200.00 | 4450.00
 Γιάννης Κορρές        | 1800.00 |        6200.00 | 4400.00
 Άγγελος Φωτίου        | 3900.00 |        8200.00 | 4300.00
 Στέλιος Παππάς        | 2900.00 |        7000.00 | 4100.00
 Κατερίνα Ζαφειρίου    | 4200.00 |        8200.00 | 4000.00
 Ιωάννα Ρήγα           | 3100.00 |        7000.00 | 3900.00
 Μαρίνα Σταθοπούλου    | 4400.00 |        8200.00 | 3800.00
 Δημήτρης Λαμπρόπουλος | 6200.00 |        9500.00 | 3300.00
 Σοφία Κωνσταντίνου    | 7000.00 |        9500.00 | 2500.00
 Πέτρος Χατζής         | 2100.00 |        4200.00 | 2100.00
 Βασιλική Ντόκου       | 1100.00 |        3100.00 | 2000.00
 Νίκος Αλεξίου         | 8200.00 |        9500.00 | 1300.00
 Χριστίνα Αθανασίου    | 1650.00 |        1800.00 |  150.00
(13 rows)
```

ΑΣΚΗΣΗ 9.3 Ζεύγη προϊόντων

Βρες ζεύγη διαφορετικών προϊόντων της ίδιας κατηγορίας που διαφέρουν στην τιμή λιγότερο από 5 ευρώ. Κάθε ζεύγος πρέπει να εμφανίζεται μία φορά. Εμφάνισε τα δύο ονόματα και τις δύο τιμές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                name                | price |       name        | price
------------------------------------+-------+-------------------+-------
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90 | Ματωμένα χώματα   | 15.20
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90 | Το Κιβώτιο       | 14.50
 Μαθαίνω Python                     | 32.00 | Ruby από την αρχή | 29.50
 Το Κιβώτιο                         | 14.50 | Ματωμένα χώματα   | 15.20
 Το Κιβώτιο                         | 14.50 | Η Φόνισσα         |  9.90
(5 rows)
```

ΑΣΚΗΣΗ 9.4 Πλέγμα καταστάσεων και μεθόδων

Φτιάξε με `CROSS JOIN` και δύο λίστες `VALUES` όλους τους συνδυασμούς των καταστάσεων `paid`, `shipped` και `delivered` με τις μεθόδους πληρωμής `card` και `cod`. Ταξινόμησε κατά κατάσταση και μέθοδο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  status   | method
-----------+--------
 delivered | card
 delivered | cod
 paid      | card
 paid      | cod
 shipped   | card
 shipped   | cod
(6 rows)
```

ΑΣΚΗΣΗ 9.5 Ηλικιακές ομάδες

Κατάταξε τους πελάτες που έχουν ημερομηνία γέννησης και `id` έως 10 σε ομάδες `έως 30`, `31 έως 45` και `46 και πάνω`, με βάση την ηλικία τους στις 30 Ιουνίου 2026. Χρησιμοποίησε join με λίστα `VALUES` και όχι `CASE`. Εμφάνισε όνομα, ηλικία και ομάδα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 first_name | years |    label
------------+-------+-------------
 Μαρία      |    38 | 31 έως 45
 Γιώργος    |    46 | 46 και πάνω
 Ελένη      |    31 | 31 έως 45
 Αναστασία  |    24 | έως 30
 Νίκος      |    36 | 31 έως 45
 Δήμητρα    |    40 | 31 έως 45
 Παναγιώτης |    54 | 46 και πάνω
 Σοφία      |    26 | έως 30
 Χρήστος    |    42 | 31 έως 45
(9 rows)
```

ΑΣΚΗΣΗ 9.6 Τρία επίπεδα ιεραρχίας

Εμφάνισε κάθε υπάλληλο που έχει προϊστάμενο και προϊστάμενο του προϊσταμένου, μαζί με τα δύο αυτά ονόματα. Ταξινόμησε κατά `id` υπαλλήλου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      employee      |        manager         |          top
--------------------+------------------------+-----------------------
 Κατερίνα Ζαφειρίου | Νίκος Αλεξίου          | Ελένη Μαυρίδου
 Άγγελος Φωτίου     | Νίκος Αλεξίου          | Ελένη Μαυρίδου
 Μαρίνα Σταθοπούλου | Νίκος Αλεξίου          | Ελένη Μαυρίδου
 Πέτρος Χατζής      | Κατερίνα Ζαφειρίου     | Νίκος Αλεξίου
 Ιωάννα Ρήγα        | Σοφία Κωνσταντίνου     | Ελένη Μαυρίδου
 Στέλιος Παππάς     | Σοφία Κωνσταντίνου     | Ελένη Μαυρίδου
 Βασιλική Ντόκου    | Ιωάννα Ρήγα            | Σοφία Κωνσταντίνου
 Γιάννης Κορρές     | Δημήτρης Λαμπρόπουλος  | Ελένη Μαυρίδου
 Θανάσης Μπέης      | Δημήτρης Λαμπρόπουλος  | Ελένη Μαυρίδου
 Χριστίνα Αθανασίου | Γιάννης Κορρές         | Δημήτρης Λαμπρόπουλος
(10 rows)
```

ΑΣΚΗΣΗ 9.7 Πληρωμή μετά την αποστολή

Βρες τις πληρωμές που έγιναν μετά την αποστολή της παραγγελίας τους και είναι θετικές. Εμφάνισε `id` παραγγελίας, μέθοδο, ημερομηνία αποστολής και ημερομηνία πληρωμής ως `date`. Κράτα όσες έγιναν από τον Απρίλιο του 2026.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | method |  shipped   |    paid
-----+--------+------------+------------
 123 | cod    | 2026-05-20 | 2026-05-21
 125 | cod    | 2026-05-25 | 2026-05-26
 128 | cod    | 2026-05-26 | 2026-05-27
 137 | cod    | 2026-06-07 | 2026-06-09
 136 | cod    | 2026-06-09 | 2026-06-10
 151 | cod    | 2026-06-18 | 2026-06-20
 146 | cod    | 2026-06-19 | 2026-06-20
(7 rows)
```

ΚΕΦΑΛΑΙΟ 10

Aggregates και GROUP BY

Μια aggregate function παίρνει πολλές γραμμές και δίνει μία τιμή. Το `GROUP BY` χωρίζει τις γραμμές σε ομάδες και το αποτέλεσμα έχει μία γραμμή ανά ομάδα. Από εδώ και πέρα κάθε query μπορεί να απαντά ερωτήσεις του τύπου «πόσα» και «πόσο».

Aggregates πάνω σε όλον τον πίνακα

Οι πέντε βασικές aggregate functions είναι οι `count`, `sum`, `avg`, `min` και `max`. Χωρίς `GROUP BY` εφαρμόζονται σε όλες τις γραμμές που πέρασαν το `WHERE` και το αποτέλεσμα έχει ακριβώς μία γραμμή.

SQL

```
SELECT count(*) AS products,
       min(price) AS cheapest,
       max(price) AS most_expensive,
       round(avg(price), 2) AS avg_price,
       sum(stock) AS units_in_stock
FROM products
WHERE active;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 products | cheapest | most_expensive | avg_price | units_in_stock
----------+----------+----------------+-----------+----------------
       29 |     9.90 |        2199.00 |    232.06 |           1566
(1 row)
```

Το `count(*)` μετράει γραμμές. Το `count(στήλη)` μετράει τις γραμμές όπου η στήλη δεν είναι `NULL`. Το `count(DISTINCT στήλη)` μετράει τις διαφορετικές τιμές που δεν είναι `NULL`. Η διαφορά είναι ουσιαστική.

SQL

```
SELECT count(*) AS customers,
       count(email) AS with_email,
       count(city_id) AS with_city,
       count(DISTINCT city_id) AS cities
FROM customers;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customers | with_email | with_city | cities
-----------+------------+-----------+--------
        30 |         28 |        28 |     10
(1 row)
```

Τα aggregates αγνοούν το NULL

Όλες οι aggregate functions εκτός από το `count(*)` αγνοούν τις τιμές `NULL`. Ο μέσος μισθός υπολογίζεται μόνο από όσους έχουν γνωστό μισθό.

SQL

```
SELECT avg(salary) AS avg_known,
       avg(COALESCE(salary, 0)) AS avg_with_zero,
       sum(salary) / count(*) AS sum_over_all
FROM employees;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       avg_known       |     avg_with_zero     |     sum_over_all
-----------------------+-----------------------+-----------------------
 4128.5714285714285714 | 3853.3333333333333333 | 3853.3333333333333333
(1 row)
```

Το `avg(salary)` διαιρεί με 14, όσους έχουν μισθό. Το `avg(COALESCE(salary, 0))` διαιρεί με 15 και μετράει τον σύμβουλο με μισθό μηδέν. Η τρίτη στήλη δίνει το ίδιο με τη δεύτερη, επειδή το `sum` αγνοεί το `NULL` ενώ το `count(*)` μετράει όλες τις γραμμές. Ποιο είναι σωστό εξαρτάται από την ερώτηση. Το κεφάλαιο 3 σε προειδοποίησε ότι το `COALESCE` μπορεί να αλλάξει έναν μέσο όρο. Εδώ βλέπεις πόσο.

Υπάρχει και η αντίστροφη περίπτωση. Όταν καμία γραμμή δεν περνά το `WHERE`, το `count` δίνει 0 αλλά το `sum` δίνει `NULL`, όχι μηδέν.

SQL

```
SELECT count(*) AS n, sum(amount) AS total, COALESCE(sum(amount), 0) AS total_or_zero
FROM payments
WHERE method = 'bitcoin';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 n | total | total_or_zero
---+-------+---------------
 0 |     ∅ |             0
(1 row)
```

GROUP BY

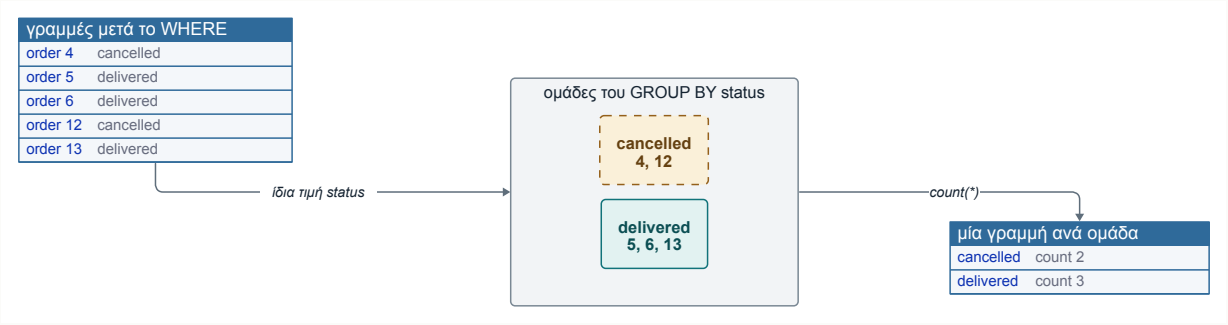
Το `GROUP BY` μαζεύει σε μία ομάδα όλες τις γραμμές με την ίδια τιμή σε μια έκφραση. Οι aggregate functions υπολογίζονται τότε ξεχωριστά για κάθε ομάδα.

SQL

```
SELECT status, count(*) AS orders
FROM orders
GROUP BY status
ORDER BY orders DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  status   | orders
-----------+--------
 delivered |    125
 cancelled |     14
 shipped   |     12
 paid      |      5
 pending   |      3
 refunded  |      3
(6 rows)
```



Σχήμα 10.1 · Το GROUP BY μαζεύει τις γραμμές με την ίδια τιμή σε ομάδες. Κάθε ομάδα γίνεται μία γραμμή του αποτελέσματος.

Μετά το `GROUP BY` δεν υπάρχουν πια μεμονωμένες γραμμές, μόνο ομάδες. Γι' αυτό η λίστα του `SELECT` μπορεί να περιέχει μόνο εκφράσεις που έχουν μία τιμή ανά ομάδα. Αυτές είναι οι εκφράσεις του `GROUP BY` και οι aggregate functions. Μια στήλη που διαφέρει μέσα στην ομάδα δεν έχει μία τιμή για να εμφανιστεί.

SQL

```
SELECT status, created_at, count(*)
FROM orders
GROUP BY status;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  column "orders.created_at" must appear in the GROUP BY clause or be used in an aggregate function
LINE 1: SELECT status, created_at, count(*)
                       ^
```

Η PostgreSQL κάνει μία εξαίρεση. Αν ομαδοποιήσεις με το primary key ενός πίνακα, ξέρει ότι κάθε άλλη στήλη του ίδιου πίνακα έχει μία τιμή ανά ομάδα. Μπορείς λοιπόν να γράψεις `GROUP BY c.id` και να εμφανίσεις το `c.first_name` χωρίς να το προσθέσεις στο `GROUP BY`.

Με πολλές εκφράσεις στο `GROUP BY`, κάθε ομάδα είναι ένας διαφορετικός συνδυασμός τιμών.

SQL

```
SELECT status, coupon_code IS NOT NULL AS used_coupon, count(*) AS orders
FROM orders
GROUP BY status, coupon_code IS NOT NULL
ORDER BY status, used_coupon;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  status   | used_coupon | orders
-----------+-------------+--------
 cancelled | f           |     12
 cancelled | t           |      2
 delivered | f           |    112
 delivered | t           |     13
 paid      | f           |      2
 paid      | t           |      3
 pending   | f           |      2
 pending   | t           |      1
 refunded  | f           |      2
 refunded  | t           |      1
 shipped   | f           |      9
 shipped   | t           |      3
(12 rows)
```

Ομαδοποίηση με έκφραση

Μπορείς να ομαδοποιήσεις με οποιαδήποτε έκφραση. Οι παραγγελίες ανά μήνα είναι `GROUP BY date_trunc('month', created_at)`.

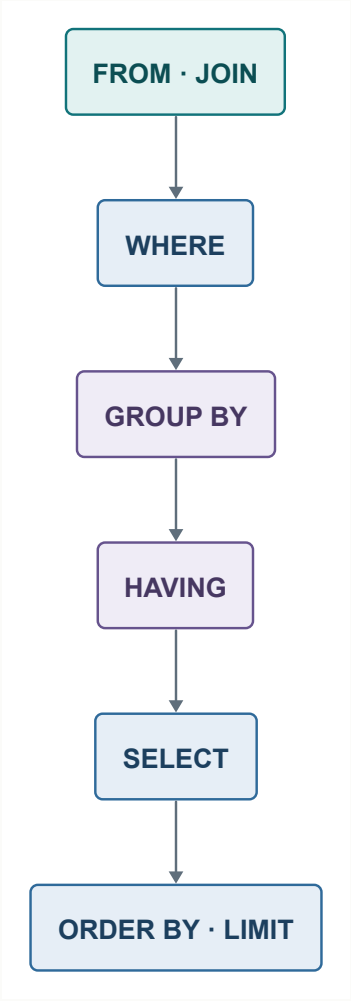
SQL

```sql
SELECT date_trunc('month', created_at)::date AS month, count(*) AS orders
FROM orders
WHERE created_at >= '2026-01-01'
GROUP BY date_trunc('month', created_at)
ORDER BY month;
```

ΑΠΟΤΕΛΕΣΜΑ

```
   month    | orders
------------+--------
 2026-01-01 |      6
 2026-02-01 |     12
 2026-03-01 |     12
 2026-04-01 |      9
 2026-05-01 |     20
 2026-06-01 |     29
(6 rows)
```

Το alias `month` επιτρέπεται στο `ORDER BY` αλλά όχι στο `GROUP BY`, αφού το `GROUP BY` εκτελείται πριν από το `SELECT`. Η PostgreSQL είναι λίγο πιο ανεκτική από το πρότυπο και δέχεται alias και εκεί, αν δεν συγκρούεται με όνομα στήλης. Εδώ το γράφουμε αναλυτικά για να είναι σαφές.



Σχήμα 10.2 · Η λογική σειρά με ομαδοποίηση. Το WHERE φιλτράρει γραμμές πριν από τις ομάδες, το HAVING φιλτράρει ομάδες.

Aggregates μετά από joins

Η αξία μιας παραγγελίας είναι το άθροισμα των γραμμών της. Στο κατάστημά μας τα κουπόνια καταγράφονται για λόγους marketing και δεν αλλάζουν τα ποσά.

```sql
SELECT oi.order_id, count(*) AS lines, sum(oi.quantity * oi.unit_price) AS total
FROM order_items AS oi
GROUP BY oi.order_id
ORDER BY total DESC
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | lines |  total
----------+-------+---------
      114 |     3 | 3047.00
       24 |     3 | 3011.00
       67 |     4 | 2717.70
       40 |     4 | 2527.50
       44 |     2 | 2448.00
(5 rows)
```

Με joins μπορείς να ομαδοποιήσεις με στήλη άλλου πίνακα. Τα έσοδα ανά κατηγορία για τις παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν είναι αυτά.

SQL

```
SELECT cat.name AS category,
       sum(oi.quantity) AS units,
       sum(oi.quantity * oi.unit_price) AS revenue
FROM order_items AS oi
JOIN orders AS o ON o.id = oi.order_id
JOIN products AS p ON p.id = oi.product_id
JOIN categories AS cat ON cat.id = p.category_id
WHERE o.status NOT IN ('cancelled', 'refunded')
GROUP BY cat.name
ORDER BY revenue DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
     category     | units | revenue
------------------+-------+----------
 Laptops          |    28 | 36572.00
 Γραφείο          |    36 |  8800.00
 Περιφερειακά     |    48 |  6675.10
 Ήχος             |    35 |  4985.00
 Κουζίνα          |    34 |  3520.90
 Προγραμματισμός  |    39 |  1292.00
 Βάσεις δεδομένων |    28 |  1125.00
 Λογοτεχνία       |    56 |   771.10
 Τεχνολογία       |    12 |   492.00
(9 rows)
```

Το `WHERE` πέταξε τις ακυρωμένες παραγγελίες πριν από την ομαδοποίηση. Η δωροκάρτα δεν έχει κατηγορία και το `INNER JOIN` με τον `categories` την άφησε έξω. Αν ήθελες να τη δεις σε μια γραμμή `χωρίς κατηγορία`, θα έγραφες `LEFT JOIN` και `COALESCE(cat.name, 'χωρίς κατηγορία')` και στο `SELECT` και στο `GROUP BY`.

Πολλές τιμές σε μία

Το `string_agg(κείμενο, διαχωριστικό)` ενώνει τα κείμενα μιας ομάδας σε ένα. Μέσα στις παρενθέσεις δέχεται δικό του `ORDER BY`, που ορίζει τη σειρά των κομματιών. Το `array_agg` κάνει το ίδιο αλλά φτιάχνει πίνακα τιμών.

SQL

```
SELECT t.name AS city,
       count(*) AS customers,
       string_agg(c.first_name, ', ' ORDER BY c.first_name) AS names
FROM customers AS c
JOIN cities AS t ON t.id = c.city_id
GROUP BY t.name
ORDER BY customers DESC, city
LIMIT 4;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    city     | customers |                              names
-------------+-----------+-----------------------------------------------------------------
 Αθήνα       |         7 | Αικατερίνη, Αλέξανδρος, Ελένη, Μαρία, Όλγα, Παναγιώτης, Στέλιος
 Θεσσαλονίκη |         6 | Βασίλης, Γιώργος, Ζωή, Νίκος, Πέτρος, Giorgos
 Ηράκλειο    |         3 | Αναστασία, Ηλίας, Φωτεινή
 Βόλος       |         2 | Άννα, Σοφία
(4 rows)
```

Τα `bool_and` και `bool_or` είναι τα aggregates για boolean. Το πρώτο είναι αληθές όταν όλες οι τιμές είναι αληθείς και το δεύτερο όταν τουλάχιστον μία.

SQL

```
SELECT category_id,
       bool_and(active) AS all_active,
       bool_or(stock = 0) AS some_out_of_stock
FROM products
WHERE category_id IN (2, 8, 9)
GROUP BY category_id
ORDER BY category_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 category_id | all_active | some_out_of_stock
-------------+------------+-------------------
           2 | t          | t
           8 | f          | t
           9 | t          | t
(3 rows)
```

ΚΡΑΤΑ ΑΥΤΟ

Τα aggregates μετατρέπουν πολλές γραμμές σε μία τιμή και αγνοούν το `NULL`, εκτός από το `count(*)`. Το `GROUP BY` φτιάχνει μία γραμμή ανά ομάδα. Στο `SELECT` μένουν μόνο οι εκφράσεις του `GROUP BY` και aggregates. Το `WHERE` φιλτράρει πριν από την ομαδοποίηση.

Ασκήσεις

ΑΣΚΗΣΗ 10.1 Στοιχεία πελατών

Σε ένα query μέτρησε τους πελάτες, πόσοι έχουν τηλέφωνο, πόσοι έχουν εγγραφεί στο newsletter και σε πόσες διαφορετικές πόλεις μένουν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 customers | with_phone | newsletter | cities
-----------+------------+------------+--------
        30 |         22 |         13 |     10
(1 row)
```

ΑΣΚΗΣΗ 10.2 Πληρωμές ανά μέθοδο

Για κάθε μέθοδο πληρωμής εμφάνισε πόσες πληρωμές έγιναν και το καθαρό σύνολο, δηλαδή το άθροισμα όλων των ποσών μαζί με τις επιστροφές. Ταξινόμησε από το μεγαλύτερο σύνολο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  method  | payments | net_total
----------+----------+-----------
 card     |       87 |  36905.50
 paypal   |       28 |   9977.40
 bank     |       15 |   8463.50
 cod      |       17 |   7336.80
 giftcard |        8 |    400.00
(5 rows)
```

ΑΣΚΗΣΗ 10.3 Τιμές ανά κατηγορία

Για κάθε κατηγορία με ενεργά προϊόντα εμφάνισε το όνομά της, πόσα ενεργά προϊόντα έχει, τη μικρότερη, τη μεγαλύτερη και τη μέση τιμή τους σε δύο δεκαδικά. Ταξινόμησε κατά όνομα κατηγορίας.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     category     | products | min_price | max_price | avg_price
------------------+----------+-----------+-----------+-----------
 Βάσεις δεδομένων |        2 |     36.00 |     45.00 |     40.50
 Γραφείο          |        3 |     42.00 |    459.00 |    263.33
 Ήχος             |        4 |     79.00 |    229.00 |    156.50
 Κουζίνα          |        3 |     34.90 |    249.00 |    115.97
 Λογοτεχνία       |        4 |      9.90 |     16.90 |     14.13
 Περιφερειακά     |        5 |     24.90 |    329.00 |    128.18
 Προγραμματισμός  |        3 |     29.50 |     38.00 |     33.17
 Τεχνολογία       |        1 |     41.00 |     41.00 |     41.00
 Laptops          |        3 |    649.00 |   2199.00 |   1332.33
(9 rows)
```

ΑΣΚΗΣΗ 10.4 Έσοδα ανά μήνα

Εμφάνισε για κάθε μήνα του 2026 τον αριθμό των παραγγελιών που δεν ακυρώθηκαν ούτε επιστράφηκαν και τα έσοδά τους. Ο μήνας να εμφανίζεται ως `YYYY-MM`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  month   | orders | revenue
----------+--------+----------
 2026-01 |      6 |  1860.80
 2026-02 |     11 |  3747.20
 2026-03 |     10 |  1599.20
 2026-04 |      8 |  4026.50
 2026-05 |     18 | 10338.80
 2026-06 |     28 |  9074.90
(6 rows)
```

ΑΣΚΗΣΗ 10.5 Ιστορικό πελάτη

Για τους πελάτες 1 έως 6 εμφάνισε `id`, όνομα, πλήθος παραγγελιών και τις ημερομηνίες της πρώτης και της τελευταίας ως `date`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | orders | first_order | last_order
----+------------+--------+-------------+------------
  1 | Μαρία      |     12 | 2025-03-18  | 2026-05-26
  2 | Γιώργος    |      9 | 2025-02-03  | 2026-05-25
  3 | Ελένη      |      8 | 2025-01-28  | 2026-06-14
  4 | Κώστας     |      8 | 2025-01-17  | 2026-06-09
  5 | Αναστασία  |      7 | 2025-02-06  | 2026-06-06
  6 | Νίκος      |     10 | 2025-01-19  | 2026-06-13
(6 rows)
```

ΑΣΚΗΣΗ 10.6 Οι μεγαλύτερες παραγγελίες

Εμφάνισε τις πέντε παραγγελίες με τη μεγαλύτερη αξία, μαζί με το όνομα του πελάτη και το πλήθος των τεμαχίων τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | first_name |  last_name   | units |  total
-----+------------+--------------+-------+---------
 114 | Στέλιος    | Κουτσούκος   |     3 | 3047.00
  24 | Γιώργος    | Παπαδόπουλος |     5 | 3011.00
  67 | Σοφία      | Αλεξίου      |     6 | 2717.70
  40 | Κατερίνα   | Σταύρου      |     4 | 2527.50
  44 | Παναγιώτης | Βασιλείου    |     2 | 2448.00
(5 rows)
```

ΑΣΚΗΣΗ 10.7 Τι περιείχε κάθε παραγγελία

Για κάθε παραγγελία του πελάτη 13 εμφάνισε το `id` και τα ονόματα των προϊόντων της σε μία στήλη, αλφαβητικά, χωρισμένα με κόμμα και κενό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |                                products
----+-------------------------------------------------------------------------
 20 | Δίκτυα υπολογιστών, Η Φόνισσα, Σετ μαχαιριών
 35 | Ασύρματο ποντίκι, Βίος και πολιτεία του Αλέξη Ζορμπά, Laptop Student 15
 40 | Δωροκάρτα 50€, Καφετιέρα espresso, Laptop Pro 16, Ruby από την αρχή
 41 | Εργονομική καρέκλα
 82 | Μικρόφωνο USB, PostgreSQL σε βάθος
(5 rows)
```

ΑΣΚΗΣΗ 10.8 Βαθμολογίες προϊόντων

Για κάθε προϊόν με κριτικές εμφάνισε όνομα, πλήθος κριτικών, μέση βαθμολογία με ένα δεκαδικό και τη χαμηλότερη βαθμολογία. Κράτα μόνο τα προϊόντα της κατηγορίας 10 και ταξινόμησε από τη μεγαλύτερη μέση βαθμολογία.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      name       | reviews | avg_rating | worst
-----------------+---------+------------+-------
 Μικρόφωνο USB   |       1 |        5.0 |     5
 Ακουστικά ANC   |       4 |        4.3 |     3
 Ηχείο Bluetooth |       1 |        4.0 |     4
(3 rows)
```

ΚΕΦΑΛΑΙΟ 11

HAVING, FILTER και η παγίδα του fan out

Το `HAVING` φιλτράρει ομάδες, όπως το `WHERE` φιλτράρει γραμμές. Το `FILTER` περιορίζει ένα aggregate σε μέρος της ομάδας. Και τα δύο είναι απλά. Το δύσκολο κομμάτι αυτού του κεφαλαίου είναι το fan out, η σιωπηλή διόγκωση των αθροισμάτων όταν ενώνεις πολλούς πίνακες πριν αθροίσεις.

HAVING

Θέλεις τους πελάτες με τουλάχιστον δέκα παραγγελίες. Το `WHERE` δεν μπορεί να το εκφράσει. Εκτελείται πριν από την ομαδοποίηση, όταν το πλήθος των παραγγελιών κάθε πελάτη δεν έχει μετρηθεί ακόμη.

SQL

```
SELECT customer_id, count(*) AS orders
FROM orders
WHERE count(*) >= 10
GROUP BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  aggregate functions are not allowed in WHERE
LINE 3: WHERE count(*) >= 10
              ^
```

Το `HAVING` εκτελείται μετά το `GROUP BY` και βλέπει τα aggregates κάθε ομάδας.

SQL

```
SELECT customer_id, count(*) AS orders
FROM orders
GROUP BY customer_id
HAVING count(*) >= 10
ORDER BY orders DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | orders
-------------+--------
           1 |     12
          12 |     10
           6 |     10
(3 rows)
```

Ένα query μπορεί να έχει και τα δύο. Το `WHERE` διαλέγει ποιες γραμμές μπαίνουν στις ομάδες και το `HAVING` ποιες ομάδες μένουν. Οι πελάτες με τουλάχιστον τρεις παραδομένες παραγγελίες μέσα στο 2026 είναι αυτοί.

SQL

```
SELECT customer_id, count(*) AS delivered_2026
FROM orders
WHERE status = 'delivered' AND created_at >= '2026-01-01'
GROUP BY customer_id
HAVING count(*) >= 3
ORDER BY delivered_2026 DESC, customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | delivered_2026
-------------+----------------
          23 |              6
           1 |              4
           4 |              4
          29 |              4
           5 |              3
           8 |              3
           9 |              3
          12 |              3
          25 |              3
(9 rows)
```

Μια συνθήκη που αφορά μεμονωμένες γραμμές μπορεί τεχνικά να γραφτεί και στο `HAVING`, αν η στήλη είναι στο `GROUP BY`. Γράψε τη στο `WHERE`. Είναι πιο καθαρό και οι γραμμές φεύγουν πριν από την ομαδοποίηση, όχι μετά.

FILTER

Συχνά θέλεις πολλές μετρήσεις της ίδιας ομάδας με διαφορετικά κριτήρια. Πόσες παραγγελίες έκανε κάθε πελάτης, πόσες παραδόθηκαν και πόσες ακυρώθηκαν. Το `FILTER (WHERE ...)` μετά από ένα aggregate το περιορίζει στις γραμμές της ομάδας που ικανοποιούν τη συνθήκη.

SQL

```
SELECT customer_id,
       count(*) AS orders,
       count(*) FILTER (WHERE status = 'delivered') AS delivered,
       count(*) FILTER (WHERE status = 'cancelled') AS cancelled
FROM orders
WHERE customer_id BETWEEN 1 AND 6
GROUP BY customer_id
ORDER BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | orders | delivered | cancelled
-------------+--------+-----------+-----------
           1 |     12 |        10 |         2
           2 |      9 |         8 |         1
           3 |      8 |         6 |         1
           4 |      8 |         7 |         0
           5 |      7 |         6 |         1
           6 |     10 |         5 |         3
(6 rows)
```

Στην άσκηση 10.1 έγραψες το ίδιο με `count(CASE WHEN ... THEN 1 END)`. Το αποτέλεσμα είναι ίδιο και η μορφή με `CASE` δουλεύει σε κάθε βάση. Το `FILTER` είναι μέρος του προτύπου αλλά δεν το υποστηρίζουν όλες οι βάσεις. Στην PostgreSQL διαβάζεται πιο εύκολα και δουλεύει με κάθε aggregate.

SQL

```
SELECT method,
       sum(amount) FILTER (WHERE amount > 0) AS charged,
       sum(amount) FILTER (WHERE amount < 0) AS refunded,
       sum(amount) AS net
FROM payments
GROUP BY method
ORDER BY method;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  method  | charged  | refunded |   net
----------+----------+----------+----------
 bank     |  8463.50 |        ∅ |  8463.50
 card     | 38095.50 | -1190.00 | 36905.50
 cod      |  7336.80 |        ∅ |  7336.80
 giftcard |   400.00 |        ∅ |   400.00
 paypal   | 10406.40 |  -429.00 |  9977.40
(5 rows)
```

Όπου δεν υπάρχει καμία επιστροφή, το `sum` με `FILTER` δίνει `NULL` και όχι μηδέν, για τον λόγο που είδες στο κεφάλαιο 10.

Fan out

Θέλεις για κάθε παραγγελία του πελάτη 13 την αξία των προϊόντων και το ποσό που πληρώθηκε. Και οι δύο πίνακες ενώνονται με τον `orders`, οπότε η πρώτη ιδέα είναι δύο joins και δύο αθροίσματα.

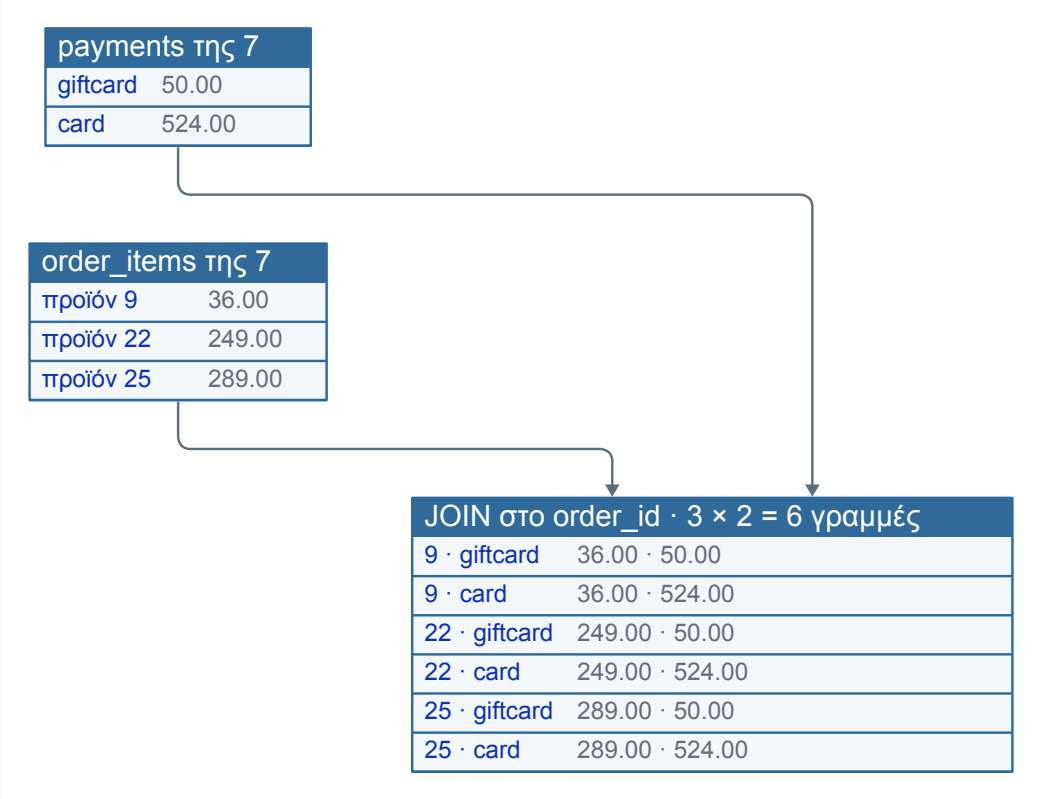
SQL

```
SELECT o.id,
       sum(oi.quantity * oi.unit_price) AS items_total,
       sum(pay.amount) AS paid_total
FROM orders AS o
JOIN order_items AS oi ON oi.order_id = o.id
JOIN payments AS pay ON pay.order_id = o.id
WHERE o.customer_id = 13
GROUP BY o.id
ORDER BY o.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | items_total | paid_total
----+-------------+------------
 20 |      114.90 |     344.70
 35 |      690.80 |    2072.40
 40 |     2527.50 |   10110.00
 41 |      578.00 |     289.00
 82 |      164.00 |     328.00
(5 rows)
```

Η παραγγελία 20 έχει τρία προϊόντα και μία πληρωμή, αλλά η πληρωμή εμφανίζεται τριπλάσια. Η 41 έχει ένα προϊόν και δύο πληρωμές, οπότε διπλασιάστηκε η αξία του προϊόντος. Καμία γραμμή δεν είναι σωστή. Το query δεν έβγαλε κανένα μήνυμα. Τα νούμερα απλώς είναι μεγαλύτερα από την αλήθεια.



Σχήμα 11.1 · Όταν ενώνεις δύο πίνακες «πολλά» με τον ίδιο «ένα», κάθε γραμμή του πρώτου ζευγαρώνει με κάθε γραμμή του δεύτερου. Τα αθροίσματα μετρούν κάθε τιμή πολλές φορές.

Είναι το φαινόμενο που είδες με το `USING` στο κεφάλαιο 7. Οι γραμμές προϊόντων και οι πληρωμές δεν έχουν σχέση μεταξύ τους. Συνδέονται μόνο μέσω της παραγγελίας, οπότε το join φτιάχνει κάθε συνδυασμό τους. Λέμε ότι το join άλλαξε το **grain** του αποτελέσματος. Πριν από το join μια γραμμή ήταν ένα προϊόν ή μια πληρωμή. Μετά είναι ένα ζεύγος προϊόντος και πληρωμής, κάτι που δεν σημαίνει τίποτα.

Διόρθωση. Άθροισε πριν ενώσεις

Η σωστή λύση είναι να αθροίσεις κάθε πίνακα στο grain της παραγγελίας και μετά να ενώσεις τα αποτελέσματα, που έχουν πια μία γραμμή ανά παραγγελία.

Για αυτό χρειάζεται ένα εργαλείο που θα δούμε αναλυτικά στο κεφάλαιο 14. Ένα ολόκληρο query μέσα σε παρενθέσεις μπορεί να σταθεί στο `FROM` ή σε ένα `JOIN` σαν να ήταν πίνακας. Του δίνεις alias και χρησιμοποιείς τις στήλες του όπως θα χρησιμοποιούσες τις στήλες ενός πίνακα.

SQL

```
SELECT o.id, it.items_total, pt.paid_total
FROM orders AS o
JOIN (SELECT order_id, sum(quantity * unit_price) AS items_total
      FROM order_items
      GROUP BY order_id) AS it ON it.order_id = o.id
JOIN (SELECT order_id, sum(amount) AS paid_total
      FROM payments
      GROUP BY order_id) AS pt ON pt.order_id = o.id
WHERE o.customer_id = 13
ORDER BY o.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | items_total | paid_total
----+-------------+------------
 20 |      114.90 |     114.90
 35 |      690.80 |     690.80
 40 |     2527.50 |    2527.50
 41 |      289.00 |     289.00
 82 |      164.00 |     164.00
(5 rows)
```

Κάθε εσωτερικό query έχει μία γραμμή ανά `order_id`. Το join τους με τον `orders` δεν μπορεί πια να πολλαπλασιάσει τίποτα. Αν κάποια παραγγελία δεν είχε πληρωμές, θα έγραφες `LEFT JOIN` στο δεύτερο και θα την κρατούσες με ∅.

Μια πιο πρόχειρη διόρθωση είναι το `count(DISTINCT ...)`. Δουλεύει για μετρήσεις, αφού οι διπλές γραμμές έχουν το ίδιο κλειδί. Για αθροίσματα δεν δουλεύει. Το `sum(DISTINCT amount)` θα έχανε δύο διαφορετικές πληρωμές που τυχαίνει να έχουν το ίδιο ποσό.

ΠΑΓΙΔΑ

Πριν γράψεις ένα aggregate μετά από joins, ρώτα τι σημαίνει μία γραμμή του αποτελέσματος. Αν ενώνεις έναν πίνακα με δύο διαφορετικούς πίνακες «πολλά», η απάντηση είναι σχεδόν πάντα «τίποτα χρήσιμο». Άθροισε κάθε πλευρά χωριστά.

ΚΡΑΤΑ ΑΥΤΟ

Το `WHERE` φιλτράρει γραμμές πριν από την ομαδοποίηση και το `HAVING` ομάδες μετά. Το `FILTER` περιορίζει ένα aggregate σε μέρος της ομάδας. Δύο joins προς πίνακες «πολλά» από τον ίδιο πίνακα φτιάχνουν καρτεσιανό γινόμενο και διογκώνουν τα αθροίσματα. Άθροισε πριν ενώσεις.

Ασκήσεις

ΑΣΚΗΣΗ 11.1 Οι πιστοί πελάτες

Εμφάνισε `id`, όνομα, επώνυμο και πλήθος παραγγελιών των πελατών με τουλάχιστον εννέα παραγγελίες, από τον πιο δραστήριο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name   | orders
----+------------+--------------+--------
  1 | Μαρία      | Παπαδοπούλου |     12
  6 | Νίκος      | Ιωάννου      |     10
 12 | Αλέξανδρος | Οικονόμου    |     10
  2 | Γιώργος    | Παπαδόπουλος |      9
(4 rows)
```

ΑΣΚΗΣΗ 11.2 Κατηγορίες με μεγάλο τζίρο

Εμφάνισε τις κατηγορίες με έσοδα πάνω από 4000 ευρώ από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν, με τα έσοδά τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   category   | revenue
--------------+----------
 Laptops      | 36572.00
 Γραφείο      |  8800.00
 Περιφερειακά |  6675.10
 Ήχος         |  4985.00
(4 rows)
```

ΑΣΚΗΣΗ 11.3 Χρεώσεις και επιστροφές

Για τους πελάτες που έχουν τουλάχιστον μία επιστροφή, εμφάνισε `customer_id`, το σύνολο των θετικών πληρωμών τους και το σύνολο των επιστροφών.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | charged | refunded
-------------+---------+----------
           4 | 3539.90 |   -90.00
           6 | 2243.50 | -1190.00
          28 | 1021.00 |  -339.00
(3 rows)
```

ΑΣΚΗΣΗ 11.4 Δημοφιλή προϊόντα

Βρες τα προϊόντα που αγόρασαν τουλάχιστον 12 διαφορετικοί πελάτες. Εμφάνισε όνομα, πλήθος πελατών και συνολικά τεμάχια.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
        name          | customers | units
----------------------+-----------+-------
 Ακουστικά ANC        |        12 |    14
 Μικρόφωνο USB        |        12 |    16
 PostgreSQL σε βάθος  |        12 |    15
(3 rows)
```

ΑΣΚΗΣΗ 11.5 Μήνες με κίνηση

Εμφάνισε τους μήνες του 2025 με τουλάχιστον 10 παραγγελίες. Για καθέναν δείξε τον μήνα ως `YYYY-MM`, το σύνολο των παραγγελιών και πόσες από αυτές είχαν κουπόνι.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  month  | orders | with_coupon
---------+--------+-------------
 2025-09 |     10 |           0
 2025-11 |     12 |           4
 2025-12 |     11 |           2
(3 rows)
```

ΑΣΚΗΣΗ 11.6 Παραγγελίες και κριτικές

Για τους πελάτες 1 έως 5 εμφάνισε `id`, πλήθος παραγγελιών και πλήθος κριτικών. Πελάτες χωρίς κριτικές πρέπει να έχουν 0. Πρόσεξε το fan out.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | orders | reviews
----+--------+---------
  1 |     12 |       3
  2 |      9 |       8
  3 |      8 |       4
  4 |      8 |       3
  5 |      7 |       2
(5 rows)
```

ΑΣΚΗΣΗ 11.7 Απλήρωτα υπόλοιπα

Βρες τις παραγγελίες με κατάσταση `delivered` όπου το ποσό που πληρώθηκε διαφέρει από την αξία των προϊόντων. Άθροισε κάθε πλευρά χωριστά πριν από το join. Αν δεν βρεις καμία, το αποτέλεσμα είναι σωστό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | items_total | paid_total
----+-------------+------------
(0 rows)
```

ΚΕΦΑΛΑΙΟ 12

UNION, INTERSECT και EXCEPT

Το join βάζει στήλες δίπλα δίπλα. Οι πράξεις συνόλων βάζουν γραμμές τη μία κάτω από την άλλη. Το `UNION` ενώνει δύο αποτελέσματα, το `INTERSECT` κρατά τις κοινές γραμμές και το `EXCEPT` όσες υπάρχουν μόνο στο πρώτο.

UNION ALL και UNION

Θέλεις μια λίστα επαφών με πελάτες και προμηθευτές. Οι δύο πίνακες έχουν διαφορετικές στήλες, αλλά και οι δύο έχουν ένα όνομα και ένα email. Γράφεις δύο queries με ίδιο πλήθος στηλών και τα ενώνεις με `UNION ALL`.

SQL

```
SELECT first_name || ' ' || last_name AS name, email
FROM customers
WHERE city_id = 4
UNION ALL
SELECT name, email
FROM suppliers
WHERE city_id = 4;
```

ΑΠΟΤΕΛΕΣΜΑ

```
        name        |          email
--------------------+------------------------
 Αναστασία Γεωργίου | ø
 Φωτεινή Λαμπράκη   | foteini.l@example.gr
 Ηλίας Τζαννετάκης  | ilias.tz@example.gr
 Κρητική Οικιακή    | info@kritiki.example.gr
(4 rows)
```

Οι κανόνες είναι τρεις. Τα δύο queries πρέπει να έχουν τον ίδιο αριθμό στηλών. Οι στήλες στην ίδια θέση πρέπει να έχουν συμβατούς τύπους. Τα ονόματα των στηλών του αποτελέσματος έρχονται από το πρώτο query.

Το `UNION ALL` κρατά όλες τις γραμμές, ακόμη και τις διπλές. Το `UNION` χωρίς `ALL` αφαιρεί τις διπλές γραμμές, όπως το `DISTINCT`.

SQL

```
SELECT city_id FROM customers WHERE city_id IN (4, 5)
UNION
SELECT city_id FROM suppliers WHERE city_id IN (4, 5);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 city_id
---------
       5
       4
(2 rows)
```

Για να αφαιρέσει διπλότυπα η βάση πρέπει να ταξινομήσει ή να κάνει hash όλο το αποτέλεσμα. Όταν ξέρεις ότι δεν υπάρχουν διπλές γραμμές ή δεν σε ενοχλούν, γράψε `UNION ALL`. Είναι φθηνότερο και λέει καθαρά τι εννοείς.

Σημάδεψε την πηγή

Σε μια ένωση συχνά θέλεις να ξέρεις από πού ήρθε κάθε γραμμή. Προσθέτεις μια σταθερή στήλη σε κάθε query.

SQL

```
SELECT 'πελάτης' AS kind, first_name || ' ' || last_name AS name, email
FROM customers
WHERE city_id = 5
UNION ALL
SELECT 'προμηθευτής', name, email
FROM suppliers
WHERE city_id = 5
UNION ALL
SELECT 'υπάλληλος', full_name, email
FROM employees
WHERE department = 'Αποθήκη'
ORDER BY kind, name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    kind     |         name          |            email
-------------+-----------------------+-----------------------------
 πελάτης     | Δήμητρα Κωνσταντίνου  | dimitra.k@example.gr
 πελάτης     | Μιχάλης Τσακίρης      | michalis.ts@example.gr
 προμηθευτής | Θεσσαλική Χαρτική     | xartika@example.gr
 υπάλληλος   | Γιάννης Κορρές        | giannis.k@shop.example.gr
 υπάλληλος   | Δημήτρης Λαμπρόπουλος | dimitris.l@shop.example.gr
 υπάλληλος   | Θανάσης Μπέης         | thanasis.m@shop.example.gr
 υπάλληλος   | Χριστίνα Αθανασίου    | christina.a@shop.example.gr
(7 rows)
```

Το `ORDER BY` στο τέλος ταξινομεί ολόκληρο το αποτέλεσμα, όχι μόνο το τελευταίο query. Αναφέρεται στα ονόματα στηλών του πρώτου query. Αν θέλεις `LIMIT` ή `ORDER BY` μόνο σε ένα από τα κομμάτια, το βάζεις σε παρενθέσεις.

INTERSECT και EXCEPT

Το `INTERSECT` κρατά τις γραμμές που υπάρχουν και στα δύο αποτελέσματα. Οι πελάτες που αγόρασαν και το 2025 και το 2026 είναι η τομή δύο λιστών.

SQL

```
SELECT customer_id FROM orders
WHERE created_at >= '2025-01-01' AND created_at < '2026-01-01'
INTERSECT
SELECT customer_id FROM orders
WHERE created_at >= '2026-01-01'
ORDER BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id
-------------
           1
           2
           3
           4
           5
           6
           7
           8
           9
          11
          12
          13
          14
          15
…
(21 rows)
```

Το `EXCEPT` κρατά τις γραμμές του πρώτου που δεν υπάρχουν στο δεύτερο. Οι πελάτες του 2025 που δεν έχουν αγοράσει τίποτα από τον Μάρτιο του 2026 είναι αυτοί.

SQL

```
SELECT customer_id FROM orders
WHERE created_at >= '2025-01-01' AND created_at < '2026-01-01'
EXCEPT
SELECT customer_id FROM orders
WHERE created_at >= '2026-03-01'
ORDER BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id
-------------
          13
          21
(2 rows)
```

Και τα δύο αφαιρούν διπλότυπα, όπως το `UNION`. Υπάρχουν και σε μορφή `ALL`, που μετρά πόσες φορές εμφανίζεται κάθε γραμμή, αλλά χρειάζονται σπάνια.

Το `EXCEPT` είναι ένας τρίτος τρόπος για anti join, μετά το `LEFT JOIN` του κεφαλαίου 8 και το `NOT EXISTS` που έρχεται στο επόμενο. Έχει έναν περιορισμό. Επιστρέφει μόνο τις στήλες που συγκρίνει. Αν θέλεις και το όνομα του πελάτη, πρέπει να κάνεις join το αποτέλεσμα ξανά με τον `customers`.

Οι πράξεις συνόλων και το NULL

Οι πράξεις συνόλων θεωρούν ίσες δύο γραμμές όταν οι τιμές τους δεν διαφέρουν. Για αυτή τη σύγκριση δύο `NULL` είναι ίδια, όπως στο `DISTINCT`.

SQL

```
SELECT city_id FROM customers WHERE city_id IS NULL
INTERSECT
SELECT city_id FROM suppliers WHERE city_id IS NULL;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 city_id
---------
       ∅
(1 row)
```

Ένα join με `ON c.city_id = s.city_id` δεν θα έβρισκε ποτέ αυτό το ζεύγος. Η διαφορά μετράει όταν συγκρίνεις δύο πίνακες για να βρεις διαφορές.

Σειρά εκτέλεσης

Το `INTERSECT` εκτελείται πριν από το `UNION` και το `EXCEPT`, όπως το `AND` πριν από το `OR`. Όταν συνδυάζεις πράξεις, βάλε παρενθέσεις.

SQL

```
(SELECT customer_id FROM orders WHERE status = 'cancelled'
 UNION
 SELECT customer_id FROM orders WHERE status = 'refunded')
INTERSECT
SELECT customer_id FROM orders WHERE created_at >= '2026-01-01'
ORDER BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id
-------------
           1
           2
           3
           4
           5
           6
           9
          11
          12
          15
          17
          25
          28
(13 rows)
```

Οι πελάτες με ακύρωση ή επιστροφή που αγόρασαν και μέσα στο 2026. Χωρίς τις παρενθέσεις, το `INTERSECT` θα εκτελούνταν πρώτο ανάμεσα στο δεύτερο και στο τρίτο query.

Ενώνοντας aggregates

Ένα συχνό μοτίβο είναι μια αναφορά με γραμμές λεπτομέρειας και μια γραμμή συνόλου στο τέλος. Τα δύο κομμάτια είναι διαφορετικά queries με ίδιες στήλες.

SQL

```
SELECT method, sum(amount) AS total, 1 AS sort_key
FROM payments
GROUP BY method
UNION ALL
SELECT 'σύνολο', sum(amount), 2
FROM payments
ORDER BY sort_key, method;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  method  |  total   | sort_key
----------+----------+----------
 bank     |  8463.50 |        1
 card     | 36905.50 |        1
 cod      |  7336.80 |        1
 giftcard |   400.00 |        1
 paypal   |  9977.40 |        1
 σύνολο   | 63083.20 |        2
(6 rows)
```

Η στήλη `sort_key` κρατά τη γραμμή του συνόλου στο τέλος. Στο κεφάλαιο 26 θα δεις το `ROLLUP`, που παράγει τέτοιες γραμμές συνόλων αυτόματα και σε ένα μόνο πέρασμα.

ΚΡΑΤΑ ΑΥΤΟ

Οι πράξεις συνόλων ενώνουν γραμμές από queries με ίδιο πλήθος και συμβατούς τύπους στηλών. Το `UNION ALL` κρατά τα πάντα. Το `UNION`, το `INTERSECT` και το `EXCEPT` αφαιρούν διπλότυπα και θεωρούν ίσα τα `NULL`. Το `ORDER BY` εφαρμόζεται στο συνολικό αποτέλεσμα.

Ασκήσεις

ΑΣΚΗΣΗ 12.1 Ενιαίος κατάλογος email

Φτιάξε μια λίστα με τα email των υπαλλήλων του τμήματος Τεχνολογία και των προμηθευτών, όπου υπάρχει email, με δεύτερη στήλη `kind` που λέει `υπάλληλος` ή `προμηθευτής`. Ταξινόμησε κατά email.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
              email               |    kind
----------------------------------+-------------
 angelos.f@shop.example.gr        | υπάλληλος
 hello@nordicoffice.example.com   | προμηθευτής
 info@kritiki.example.gr          | προμηθευτής
 katerina.z@shop.example.gr       | υπάλληλος
 marina.s@shop.example.gr         | υπάλληλος
 nikos.a@shop.example.gr          | υπάλληλος
 orders@vivliodianomi.example.gr  | προμηθευτής
 petros.ch@shop.example.gr        | υπάλληλος
 sales@technoimport.example.gr    | προμηθευτής
 xartika@example.gr               | προμηθευτής
(10 rows)
```

ΑΣΚΗΣΗ 12.2 Πόλεις με πελάτες ή προμηθευτές

Εμφάνισε χωρίς επαναλήψεις τα ονόματα των πόλεων όπου μένει τουλάχιστον ένας πελάτης ή έχει έδρα τουλάχιστον ένας προμηθευτής, ταξινομημένα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    name
-------------
 Αθήνα
 Βόλος
 Ηράκλειο
 Θεσσαλονίκη
 Ιωάννινα
 Καλαμάτα
 Λάρισα
 Πάτρα
 Ρόδος
 Χανιά
(10 rows)
```

ΑΣΚΗΣΗ 12.3 Προϊόντα του 2025 που δεν ξαναπουλήθηκαν

Βρες τα `product_id` που πουλήθηκαν σε παραγγελίες του 2025 αλλά σε καμία παραγγελία του 2026.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 product_id
------------
         29
(1 row)
```

ΑΣΚΗΣΗ 12.4 Μέθοδοι πληρωμής και στα δύο έτη

Βρες τις μεθόδους πληρωμής που χρησιμοποιήθηκαν και τον Δεκέμβριο του 2025 και τον Ιούνιο του 2026.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 method
--------
 bank
 card
 cod
 paypal
(4 rows)
```

ΑΣΚΗΣΗ 12.5 Πελάτες που χάθηκαν, με όνομα

Εμφάνισε `id`, όνομα και επώνυμο των πελατών που αγόρασαν το δεύτερο εξάμηνο του 2025 αλλά όχι τους πέντε πρώτους μήνες του 2026. Υπολόγισε τη λίστα των `id` με `EXCEPT` και κάνε join με τον `customers`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name
----+------------+-------------
 17 | Φωτεινή    | Λαμπράκη
 18 | Σπύρος     | Μαρκόπουλος
(2 rows)
```

ΑΣΚΗΣΗ 12.6 Αναφορά με γραμμή συνόλου

Εμφάνισε το πλήθος των παραγγελιών ανά κατάσταση και στο τέλος μια γραμμή `σύνολο` με όλες τις παραγγελίες. Οι καταστάσεις να είναι ταξινομημένες αλφαβητικά πάνω από το σύνολο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  status   | orders | sort_key
-----------+--------+----------
 cancelled |     14 |        1
 delivered |    125 |        1
 paid      |      5 |        1
 pending   |      3 |        1
 refunded  |      3 |        1
 shipped   |     12 |        1
 σύνολο    |    162 |        2
(7 rows)
```

ΚΕΦΑΛΑΙΟ 13

Subqueries με IN και EXISTS

Ένα subquery είναι ένα query μέσα σε άλλο. Μπορεί να δώσει μία τιμή, μια λίστα για το `IN` ή μια απάντηση ναι ή όχι για το `EXISTS`. Το `NOT EXISTS` είναι ο πιο ασφαλής τρόπος να ρωτήσεις τι λείπει.

Scalar subquery

Ποια προϊόντα κοστίζουν περισσότερο από τη μέση τιμή; Χρειάζεσαι πρώτα τη μέση τιμή και μετά τη σύγκριση. Ένα subquery σε παρενθέσεις που επιστρέφει μία γραμμή και μία στήλη λέγεται **scalar subquery** και μπαίνει όπου μπαίνει μια τιμή.

SQL

```
SELECT name, price
FROM products
WHERE active
  AND price > (SELECT avg(price) FROM products WHERE active)
ORDER BY price DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name            |  price
----------------------------+---------
 Laptop Pro 16              | 2199.00
 Laptop Aero 14             | 1149.00
 Laptop Student 15          |  649.00
 Γραφείο ρυθμιζόμενου ύψους |  459.00
 Οθόνη 27" 4K               |  329.00
 Εργονομική καρέκλα         |  289.00
 Καφετιέρα espresso         |  249.00
(7 rows)
```

Το εσωτερικό query εκτελείται μία φορά και το αποτέλεσμά του λειτουργεί σαν σταθερά. Μπορεί να σταθεί και στη λίστα του `SELECT`.

SQL

```
SELECT name, price,
       round(price - (SELECT avg(price) FROM products WHERE active), 2) AS vs_avg
FROM products
WHERE category_id = 9
ORDER BY price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name           | price  | vs_avg
------------------------+--------+---------
 Ασύρματο ποντίκι       |  24.90 | -207.16
 Webcam HD              |  59.00 | -173.06
 Μηχανικό πληκτρολόγιο  |  89.00 | -143.06
 USB-C docking station  | 139.00 |  -93.06
 Οθόνη 27" 4K           | 329.00 |   96.94
(5 rows)
```

Ένα scalar subquery πρέπει να επιστρέφει το πολύ μία γραμμή. Αν δεν επιστρέψει καμία, η τιμή του είναι `NULL`. Αν επιστρέψει δύο, το query αποτυγχάνει την ώρα της εκτέλεσης.

SQL

```
SELECT name
FROM products
WHERE category_id = (SELECT id FROM categories WHERE parent_id = 3);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  more than one row returned by a subquery used as an expression
```

Η κατηγορία Τεχνολογία έχει δύο υποκατηγορίες, οπότε η σύγκριση με `=` δεν ορίζεται. Αυτό που εννοούσες είναι το `IN`.

IN με subquery

Το `IN` δέχεται και subquery αντί για λίστα τιμών. Το subquery επιστρέφει μία στήλη και όσες γραμμές θέλει.

SQL

```
SELECT name
FROM products
WHERE category_id IN (SELECT id FROM categories WHERE parent_id = 3)
ORDER BY name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name
----------------------------
 Καθαρός κώδικας στην πράξη
 Μαθαίνω Python
 Σχεδίαση βάσεων δεδομένων
 PostgreSQL σε βάθος
 Ruby από την αρχή
(5 rows)
```

Οι πελάτες που αγόρασαν το βιβλίο `Σχεδίαση βάσεων δεδομένων` είναι οι πελάτες των οποίων το `id` βρίσκεται στις παραγγελίες που περιέχουν το προϊόν 9.

SQL

```
SELECT id, first_name, last_name
FROM customers
WHERE id IN (SELECT o.customer_id
             FROM orders AS o
             JOIN order_items AS oi ON oi.order_id = o.id
             WHERE oi.product_id = 9)
ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name
----+------------+--------------
  2 | Γιώργος    | Παπαδόπουλος
  5 | Αναστασία  | Γεωργίου
  6 | Νίκος      | Ιωάννου
  7 | Δήμητρα    | Κωνσταντίνου
 12 | Αλέξανδρος | Οικονόμου
 18 | Σπύρος     | Μαρκόπουλος
 19 | Αικατερίνη | Ζαχαρίου
 25 | Λευτέρης   | Φραγκιαδάκης
 28 | Δέσποινα   | Μαυρίδου
 29 | Ηλίας      | Τζαννετάκης
(10 rows)
```

Με `JOIN` το ίδιο θα χρειαζόταν `DISTINCT`, επειδή ο πελάτης 2 αγόρασε το βιβλίο σε δύο παραγγελίες και θα εμφανιζόταν δύο φορές. Το `IN` ρωτά μόνο αν υπάρχει ταίρι, όχι πόσα. Αυτό λέγεται **semi join**. Κάθε γραμμή του εξωτερικού query εμφανίζεται το πολύ μία φορά.

EXISTS

Το `EXISTS (subquery)` είναι αληθές όταν το subquery επιστρέφει τουλάχιστον μία γραμμή. Το subquery συνήθως αναφέρεται σε στήλη του εξωτερικού query. Τότε λέγεται **correlated** και αξιολογείται λογικά μία φορά για κάθε εξωτερική γραμμή.

SQL

```
SELECT c.id, c.first_name, c.last_name
FROM customers AS c
WHERE EXISTS (SELECT 1
              FROM orders AS o
              WHERE o.customer_id = c.id
                AND o.coupon_code = 'BLACKFRIDAY')
ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  last_name
----+------------+--------------
  1 | Μαρία      | Παπαδοπούλου
 16 | Θανάσης    | Καραγιάννης
 21 | Giorgos    | Papadopoulos
(3 rows)
```

Το `c.id` μέσα στο subquery είναι ο πελάτης της εξωτερικής γραμμής που εξετάζεται. Το `SELECT 1` είναι σύμβαση. Το `EXISTS` δεν κοιτάζει τι επιστρέφει το subquery, μόνο αν επιστρέφει κάτι. Θα δεις και `SELECT *`, που έχει το ίδιο αποτέλεσμα.

Το «αξιολογείται μία φορά για κάθε γραμμή» είναι η λογική σημασία. Στην πράξη ο planner μετατρέπει συνήθως το `EXISTS` σε semi join και το εκτελεί μαζί για όλες τις γραμμές. Το κεφάλαιο 33 δείχνει πώς φαίνεται αυτό στο plan.

NOT EXISTS

Το `NOT EXISTS` βρίσκει τις γραμμές που δεν έχουν ταίρι. Είναι το anti join του κεφαλαίου 8 γραμμένο όπως το λες.

SQL

```
SELECT c.id, c.first_name, c.last_name
FROM customers AS c
WHERE NOT EXISTS (SELECT 1 FROM orders AS o WHERE o.customer_id = c.id)
ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | last_name
----+------------+-----------
 10 | Χρήστος    | Μιχαηλίδης
 26 | Ραλλού     | Κοντού
 30 | Όλγα       | Βλάχου
(3 rows)
```

Το αποτέλεσμα είναι ίδιο με το `LEFT JOIN ... WHERE o.id IS NULL`. Η PostgreSQL εκτελεί συνήθως και τα δύο με τον ίδιο τρόπο. Το `NOT EXISTS` διαβάζεται πιο εύκολα και δεν απαιτεί να διαλέξεις σωστά τη στήλη του ελέγχου `IS NULL`.

Η παγίδα του NOT IN

Το κεφάλαιο 3 σε προειδοποίησε ότι το `NOT IN` με λίστα που περιέχει `NULL` δεν επιστρέφει ποτέ τίποτα. Εδώ είναι η ίδια παγίδα σε μορφή που θα συναντήσεις. Ψάχνεις τις πόλεις χωρίς προμηθευτή.

SQL

```
SELECT name
FROM cities
WHERE id NOT IN (SELECT city_id FROM suppliers);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 name
------
(0 rows)
```

Το αποτέλεσμα είναι άδειο, αν και υπάρχουν επτά πόλεις χωρίς προμηθευτή. Ο προμηθευτής `Nordic Office` έχει `city_id` `NULL`. Για κάθε πόλη, το `id NOT IN (1, 2, 4, 5, NULL)` περιέχει τη σύγκριση `id <> NULL`,

που είναι άγνωστη. Η συνθήκη δεν γίνεται ποτέ αληθής. Το `NOT EXISTS` δεν έχει αυτό το πρόβλημα, επειδή η γραμμή με `NULL` απλώς δεν ταιριάζει με καμία πόλη.

SQL

```
SELECT t.name
FROM cities AS t
WHERE NOT EXISTS (SELECT 1 FROM suppliers AS s WHERE s.city_id = t.id)
ORDER BY t.name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
   name
----------
 Βόλος
 Ιωάννινα
 Καλαμάτα
 Κοζάνη
 Πάτρα
 Ρόδος
 Χανιά
(7 rows)
```

ΠΑΓΙΔΑ

Μη γράφεις `NOT IN (subquery)`. Ακόμη κι αν σήμερα η στήλη δεν έχει `NULL`, αρκεί μία γραμμή αύριο για να αδειάσει το αποτέλεσμα χωρίς κανένα μήνυμα. Γράψε `NOT EXISTS`. Το `NOT IN` με σταθερή λίστα που γράφεις εσύ είναι ασφαλές.

ANY και ALL

Το `x > ANY (subquery)` είναι αληθές όταν το `x` είναι μεγαλύτερο από τουλάχιστον μία τιμή του subquery. Το `x > ALL (subquery)` όταν είναι μεγαλύτερο από όλες. Το `= ANY` είναι ισοδύναμο με το `IN`.

SQL

```
SELECT name, price
FROM products
WHERE price > ALL (SELECT price FROM products WHERE category_id = 9)
ORDER BY price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name            |  price
----------------------------+---------
 Γραφείο ρυθμιζόμενου ύψους |  459.00
 Laptop Student 15          |  649.00
 Laptop Classic 13          |  799.00
 Laptop Aero 14             | 1149.00
 Laptop Pro 16              | 2199.00
(5 rows)
```

Τα προϊόντα που κοστίζουν περισσότερο από κάθε περιφερειακό είναι όσα κοστίζουν πάνω από την ακριβότερη οθόνη. Το ίδιο γράφεται με `price > (SELECT max(price) ...)`, που είναι συνήθως πιο καθαρό. Πρόσεξε ότι το `ALL` πάνω σε κενό subquery είναι πάντα αληθές.

Subquery μέσα σε subquery

Ένα subquery μπορεί να περιέχει aggregates, joins και άλλα subqueries. Η μέση αξία παραγγελίας χρειάζεται δύο επίπεδα. Πρώτα την αξία κάθε παραγγελίας και μετά τον μέσο όρο τους.

SQL

```
SELECT round(avg(total), 2) AS avg_order_value
FROM (SELECT order_id, sum(quantity * unit_price) AS total
      FROM order_items
      GROUP BY order_id) AS t;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 avg_order_value
-----------------
          433.42
(1 row)
```

Το `avg(sum(...))` δεν επιτρέπεται. Δεν μπορείς να βάλεις ένα aggregate μέσα σε άλλο στο ίδιο επίπεδο. Γι' αυτό το εσωτερικό query κάνει το πρώτο βήμα και το εξωτερικό το δεύτερο.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το scalar subquery δίνει μία τιμή. Το `IN (subquery)` και το `EXISTS` κάνουν semi join και δεν πολλαπλασιάζουν γραμμές. Το `NOT EXISTS` κάνει anti join και είναι ασφαλές με `NULL`. Το `NOT IN (subquery)` δεν είναι.

Ασκήσεις

ΑΣΚΗΣΗ 13.1 Πάνω από τον μέσο μισθό

Εμφάνισε όνομα και μισθό των υπαλλήλων που αμείβονται περισσότερο από τον μέσο μισθό όλων των υπαλλήλων με γνωστό μισθό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       full_name       | salary
-----------------------+---------
 Ελένη Μαυρίδου        | 9500.00
 Νίκος Αλεξίου         | 8200.00
 Σοφία Κωνσταντίνου    | 7000.00
 Δημήτρης Λαμπρόπουλος | 6200.00
 Μαρίνα Σταθοπούλου    | 4400.00
 Κατερίνα Ζαφειρίου    | 4200.00
(6 rows)
```

ΑΣΚΗΣΗ 13.2 Όσοι έκριναν τα ακουστικά

Εμφάνισε `id` και όνομα των πελατών που έγραψαν κριτική για το προϊόν με όνομα `Ακουστικά ANC`. Χρησιμοποίησε `IN` και όχι το `id` του προϊόντος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name
----+------------
  1 | Μαρία
  6 | Νίκος
  7 | Δήμητρα
  8 | Παναγιώτης
(4 rows)
```

ΑΣΚΗΣΗ 13.3 Πελάτες με επιστροφή

Με `EXISTS`, βρες τους πελάτες που έχουν τουλάχιστον μία παραγγελία `refunded`. Εμφάνισε `id`, όνομα και επώνυμο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | last_name
----+------------+-----------
  4 | Κώστας     | Δημητρίου
  6 | Νίκος      | Ιωάννου
 28 | Δέσποινα   | Μαυρίδου
(3 rows)
```

ΑΣΚΗΣΗ 13.4 Πελάτες χωρίς κριτική

Με `NOT EXISTS`, βρες τους πελάτες με `id` έως 12 που δεν έχουν γράψει καμία κριτική.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name
----+------------
 10 | Χρήστος
 11 | Ιωάννα
(2 rows)
```

ΑΣΚΗΣΗ 13.5 Πόλεις χωρίς προμηθευτή, σωστά

Εμφάνισε `id` και όνομα των πόλεων στις οποίες δεν έχει έδρα κανένας προμηθευτής αλλά μένει τουλάχιστον ένας πελάτης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |   name
----+----------
  3 | Πάτρα
  6 | Βόλος
  7 | Ιωάννινα
  8 | Χανιά
  9 | Καλαμάτα
 10 | Ρόδος
(6 rows)
```

ΑΣΚΗΣΗ 13.6 Οι παραγγελίες της τελευταίας ημέρας

Εμφάνισε `id`, `customer_id` και `created_at` των παραγγελιών που έγιναν την ίδια ημερολογιακή ημέρα με την πιο πρόσφατη παραγγελία.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  | customer_id |       created_at
-----+-------------+------------------------
 162 |          12 | 2026-06-28 17:18:00+03
(1 row)
```

ΑΣΚΗΣΗ 13.7 Ακριβότερα από κάθε είδος κουζίνας

Με `ALL`, βρες τα ενεργά προϊόντα που κοστίζουν περισσότερο από κάθε προϊόν της κατηγορίας Κουζίνα (12). Εμφάνισε όνομα και τιμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             |  price
-----------------------------+---------
 Εργονομική καρέκλα          |  289.00
 Οθόνη 27" 4K                |  329.00
 Γραφείο ρυθμιζόμενου ύψους |  459.00
 Laptop Student 15           |  649.00
 Laptop Aero 14              | 1149.00
 Laptop Pro 16               | 2199.00
(6 rows)
```

ΑΣΚΗΣΗ 13.8 Παραγγελίες πάνω από τον μέσο όρο

Εμφάνισε `id` και αξία των παραγγελιών του 2026 των οποίων η αξία ξεπερνά τη μέση αξία όλων των παραγγελιών του καταστήματος κατά περισσότερο από 1000 ευρώ.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 order_id |  total
----------+---------
      114 | 3047.00
      130 | 2297.90
      110 | 2208.90
      141 | 2199.00
(4 rows)
```

ΚΕΦΑΛΑΙΟ 14

Correlated subqueries, derived tables και LATERAL

Ένα subquery μπορεί να υπολογίζει κάτι διαφορετικό για κάθε γραμμή. Ένα query στο `FROM` λειτουργεί σαν πίνακας με δικές του στήλες. Το `LATERAL` ενώνει τις δύο ιδέες και επιτρέπει σε ένα query του `FROM` να βλέπει τις γραμμές που ήρθαν πριν από αυτό.

Correlated scalar subquery

Στο κεφάλαιο 13 το `EXISTS` αναφερόταν σε στήλη του εξωτερικού query. Το ίδιο μπορεί να κάνει ένα scalar subquery στη λίστα του `SELECT`. Για κάθε πελάτη υπολογίζει κάτι που αφορά μόνο αυτόν.

SQL

```
SELECT c.id, c.first_name,
       (SELECT count(*) FROM orders AS o WHERE o.customer_id = c.id) AS orders,
       (SELECT max(o.created_at)::date FROM orders AS o WHERE o.customer_id = c.id) AS last_order
FROM customers AS c
WHERE c.id BETWEEN 8 AND 11
ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | orders | last_order
----+------------+--------+------------
  8 | Παναγιώτης |      7 | 2026-06-15
  9 | Σοφία      |      8 | 2026-06-26
 10 | Χρήστος    |      0 | ∅
 11 | Ιωάννα     |      7 | 2026-06-23
(4 rows)
```

Ο πελάτης 10 δεν έχει παραγγελίες. Το `count(*)` του subquery δίνει 0 και το `max` δίνει `NULL`, όπως στο κεφάλαιο 10. Σε σύγκριση με το `LEFT JOIN` και `GROUP BY`, εδώ δεν υπάρχει κίνδυνος fan out. Κάθε subquery έχει τη δική του ομαδοποίηση και δεν πολλαπλασιάζει τις γραμμές του εξωτερικού query.

Το τίμημα είναι ότι κάθε subquery διαβάζει τον πίνακα ξεχωριστά. Για λίγες γραμμές δεν μετράει. Για χιλιάδες γραμμές και πολλά subqueries πάνω στον ίδιο πίνακα, ένα join με προκαταρκτικό aggregate είναι συνήθως γρηγορότερο.

Correlated subquery στο WHERE

Τα προϊόντα που κοστίζουν περισσότερο από τον μέσο όρο της κατηγορίας τους χρειάζονται διαφορετικό μέσο όρο για κάθε κατηγορία. Το subquery συγκρίνει με τον μέσο όρο της κατηγορίας της τρέχουσας γραμμής.

SQL

```
SELECT p.name, p.category_id, p.price
FROM products AS p
WHERE p.price > (SELECT avg(p2.price)
                 FROM products AS p2
                 WHERE p2.category_id = p.category_id)
ORDER BY p.category_id, p.price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                | category_id |  price
------------------------------------+-------------+---------
 Το Κιβώτιο                         |           2 |   14.50
 Ματωμένα χώματα                    |           2 |   15.20
 Βίος και πολιτεία του Αλέξη Ζορμπά |           2 |   16.90
 Καθαρός κώδικας στην πράξη         |           4 |   38.00
 PostgreSQL σε βάθος                |           5 |   45.00
 Laptop Pro 16                      |           8 | 2199.00
 USB-C docking station              |           9 |  139.00
 Οθόνη 27" 4K                       |           9 |  329.00
 Ακουστικά ANC                      |          10 |  199.00
 Πικάπ                              |          10 |  229.00
 Καφετιέρα espresso                 |          12 |  249.00
 Εργονομική καρέκλα                 |          13 |  289.00
 Γραφείο ρυθμιζόμενου ύψους         |          13 |  459.00
(13 rows)
```

Ο ίδιος πίνακας εμφανίζεται δύο φορές με διαφορετικά aliases, όπως στο self join. Το `p2.category_id = p.category_id` είναι η συσχέτιση. Χωρίς το alias `p2` η PostgreSQL δεν θα ήξερε ποιο `category_id` εννοείς.

Derived tables

Ένα subquery στο `FROM` λέγεται **derived table**. Το χρησιμοποίησες στο κεφάλαιο 11 για να αθροίσεις πριν από ένα join. Έχει και μια πιο απλή χρήση. Δίνει όνομα σε μια υπολογισμένη στήλη, ώστε να τη χρησιμοποιήσεις στο `WHERE` ή στο `GROUP BY` του εξωτερικού query.

SQL

```
SELECT name, margin_pct
FROM (SELECT name, round((price - cost) * 100 / price, 1) AS margin_pct
      FROM products) AS m
WHERE margin_pct >= 50
ORDER BY margin_pct DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
          name          | margin_pct
------------------------+------------
 Η Φόνισσα              |       57.6
 Φωτιστικό γραφείου LED |       57.1
 Βραστήρας              |       57.0
 Σετ μαχαιριών          |       56.3
 Ασύρματο ποντίκι       |       55.8
(5 rows)
```

Στο κεφάλαιο 2 είδες ότι το `WHERE` δεν βλέπει τα aliases του ίδιου `SELECT`. Εδώ το `margin_pct` υπολογίστηκε μέσα στο derived table, οπότε για το εξωτερικό query είναι μια κανονική στήλη.

Δώσε πάντα alias σε ένα derived table. Η PostgreSQL το δέχεται χωρίς alias από την έκδοση 16, αλλά οι παλαιότερες εκδόσεις και οι περισσότερες άλλες βάσεις όχι.

Ένα derived table είναι επίσης ο τρόπος να ομαδοποιήσεις σε δύο επίπεδα. Εδώ οι πελάτες χωρίζονται σε κατηγορίες ανάλογα με το πόσες παραγγελίες έχουν και μετά μετράμε κάθε κατηγορία.

SQL

```
SELECT segment, count(*) AS customers
FROM (SELECT customer_id,
             CASE WHEN count(*) >= 8 THEN 'πιστός'
                  WHEN count(*) >= 4 THEN 'τακτικός'
                  ELSE 'περιστασιακός' END AS segment
      FROM orders
      GROUP BY customer_id) AS s
GROUP BY segment
ORDER BY customers DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    segment    | customers
---------------+-----------
 τακτικός      |        12
 πιστός        |         8
 περιστασιακός |         7
(3 rows)
```

LATERAL

Ένα κανονικό derived table δεν βλέπει τους άλλους πίνακες του `FROM`. Υπολογίζεται μία φορά, ανεξάρτητα από αυτούς. Αν γράψεις `LATERAL` μπροστά του, μπορεί να αναφέρεται σε στήλες των πινάκων που γράφτηκαν πριν από αυτό. Τότε υπολογίζεται λογικά μία φορά για κάθε γραμμή τους, όπως ένα correlated subquery. Η διαφορά είναι ότι μπορεί να επιστρέψει πολλές γραμμές και πολλές στήλες.

Η κλασική χρήση είναι το top N ανά ομάδα. Οι δύο πιο πρόσφατες παραγγελίες κάθε πελάτη της Κρήτης είναι αυτές.

SQL

```
SELECT c.id, c.first_name, recent.id AS order_id, recent.created_at::date
FROM customers AS c
JOIN cities AS t ON t.id = c.city_id
CROSS JOIN LATERAL (SELECT o.id, o.created_at
                    FROM orders AS o
                    WHERE o.customer_id = c.id
                    ORDER BY o.created_at DESC
                    LIMIT 2) AS recent
WHERE t.region = 'Κρήτη'
ORDER BY c.id, recent.created_at DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name | order_id | created_at
----+------------+----------+------------
  5 | Αναστασία  |      137 | 2026-06-06
  5 | Αναστασία  |       90 | 2026-02-23
 13 | Κατερίνα   |       82 | 2026-02-04
 13 | Κατερίνα   |       41 | 2025-09-21
 17 | Φωτεινή    |      154 | 2026-06-21
 17 | Φωτεινή    |      138 | 2026-06-08
 25 | Λευτέρης   |      157 | 2026-06-23
 25 | Λευτέρης   |      140 | 2026-06-09
 29 | Ηλίας      |      159 | 2026-06-24
 29 | Ηλίας      |      153 | 2026-06-19
(10 rows)
```

Για κάθε πελάτη το subquery ταξινομεί τις παραγγελίες του και κρατά δύο. Το `CROSS JOIN LATERAL` ενώνει κάθε πελάτη με τις γραμμές που επέστρεψε το δικό του subquery. Ένας πελάτης χωρίς παραγγελίες δεν θα είχε γραμμές και θα χανόταν, όπως σε `INNER JOIN`. Για να τον κρατήσεις γράφεις `LEFT JOIN LATERAL (...) AS recent ON true`. Η συνθήκη `ON true` λέει ότι κάθε γραμμή του subquery ταιριάζει, αφού το φιλτράρισμα έγινε ήδη μέσα του.

Μια δεύτερη χρήση του `LATERAL` είναι να υπολογίσεις μια έκφραση μία φορά και να τη χρησιμοποιήσεις σε πολλά σημεία.

SQL

```
SELECT p.name, m.margin, m.margin_pct
FROM products AS p
CROSS JOIN LATERAL (SELECT p.price - p.cost AS margin,
                          round((p.price - p.cost) * 100 / p.price, 1) AS margin_pct) AS m
WHERE m.margin > 200
ORDER BY m.margin DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
      name      | margin | margin_pct
----------------+--------+------------
 Laptop Pro 16  | 479.00 |       21.8
 Laptop Aero 14 | 259.00 |       22.5
(2 rows)
```

Το `SELECT` μέσα στο `LATERAL` δεν έχει `FROM`. Υπολογίζει εκφράσεις πάνω στη γραμμή του `p` και επιστρέφει μία γραμμή με δύο στήλες.

Τρεις τρόποι για το ίδιο πρόβλημα

Η πιο πρόσφατη παραγγελία κάθε πελάτη μπορεί να γραφτεί τουλάχιστον με τρεις τρόπους που έχεις ήδη δει. Το `DISTINCT ON` του κεφαλαίου 4 είναι το πιο σύντομο. Το correlated subquery είναι το πιο φορητό σε άλλες βάσεις. Το `LATERAL` είναι το πιο ευέλικτο, γιατί αλλάζεις το `LIMIT 1` σε `LIMIT 3` και παίρνεις τις τρεις πιο πρόσφατες.

SQL

```
SELECT c.id, last_o.id AS order_id, last_o.created_at::date
FROM customers AS c
JOIN LATERAL (SELECT o.id, o.created_at
              FROM orders AS o
              WHERE o.customer_id = c.id
              ORDER BY o.created_at DESC
              LIMIT 1) AS last_o ON true
WHERE c.id <= 4
ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | order_id | created_at
----+----------+------------
  1 |      129 | 2026-05-26
  2 |      128 | 2026-05-25
  3 |      145 | 2026-06-14
  4 |      141 | 2026-06-09
(4 rows)
```

Στο κεφάλαιο 23 θα προστεθεί ένας τέταρτος, με window functions. Στο κεφάλαιο 36 θα δεις πώς ένα index κάνει το LATERAL το πιο γρήγορο από όλους για μεγάλους πίνακες.

ΚΡΑΤΑ ΑΥΤΟ

Ένα correlated subquery αναφέρεται στην εξωτερική γραμμή και υπολογίζεται λογικά για καθεμία. Ένα derived table δίνει όνομα σε υπολογισμένες στήλες και επιτρέπει ομαδοποίηση σε δύο επίπεδα. Το LATERAL είναι derived table που βλέπει τις προηγούμενες γραμμές του FROM. Είναι το εργαλείο για top N ανά ομάδα.

Ασκήσεις

ΑΣΚΗΣΗ 14.1 Προϊόντα ανά κατηγορία

Για κάθε κατηγορία εμφάνισε το όνομά της και με correlated subquery το πλήθος των ενεργών προϊόντων της. Οι κατηγορίες χωρίς προϊόντα πρέπει να έχουν 0. Ταξινόμησε κατά `id`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |       name        | products
----+-------------------+----------
  1 | Βιβλία            |        0
  2 | Λογοτεχνία        |        4
  3 | Τεχνολογία        |        1
  4 | Προγραμματισμός   |        3
  5 | Βάσεις δεδομένων  |        2
  6 | Ηλεκτρονικά       |        0
  7 | Υπολογιστές       |        0
  8 | Laptops           |        3
  9 | Περιφερειακά      |        5
 10 | Ήχος              |        4
 11 | Σπίτι             |        0
 12 | Κουζίνα           |        3
 13 | Γραφείο           |        3
(13 rows)
```

ΑΣΚΗΣΗ 14.2 Φθηνότερα από τον μέσο όρο της κατηγορίας

Εμφάνισε όνομα, κατηγορία και τιμή των ενεργών προϊόντων που κοστίζουν λιγότερο από τον μέσο όρο των ενεργών προϊόντων της κατηγορίας τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
           name             | category_id |  price
----------------------------+-------------+---------
 Η Φόνισσα                  |           2 |    9.90
 Ruby από την αρχή          |           4 |   29.50
 Μαθαίνω Python             |           4 |   32.00
 Σχεδίαση βάσεων δεδομένων |           5 |   36.00
 Laptop Student 15          |           8 |  649.00
 Laptop Aero 14             |           8 | 1149.00
 Ασύρματο ποντίκι           |           9 |   24.90
 Webcam HD                  |           9 |   59.00
 Μηχανικό πληκτρολόγιο      |           9 |   89.00
 Ηχείο Bluetooth            |          10 |   79.00
 Μικρόφωνο USB              |          10 |  119.00
 Βραστήρας                  |          12 |   34.90
 Σετ μαχαιριών              |          12 |   64.00
 Φωτιστικό γραφείου LED     |          13 |   42.00
(14 rows)
```

ΑΣΚΗΣΗ 14.3 Κάρτα πελάτη

Για τους πελάτες 25 έως 30 εμφάνισε `id`, όνομα, το συνολικό καθαρό ποσό που έχουν πληρώσει και την ημερομηνία της τελευταίας πληρωμής. Όσοι δεν έχουν πληρώσει πρέπει να έχουν 0 και ∅.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | first_name |  paid   | last_payment
----+------------+---------+--------------
 25 | Λευτέρης   | 1424.90 | 2026-06-09
 26 | Ραλλού     |       0 | ∅
 27 | Πέτρος     |  557.80 | 2026-06-25
 28 | Δέσποινα   |  682.00 | 2026-06-17
 29 | Ηλίας      |  850.40 | 2026-06-19
 30 | Όλγα       |       0 | ∅
(6 rows)
```

ΑΣΚΗΣΗ 14.4 Ηλικιακές ομάδες, χωρίς επανάληψη

Ξαναγράψε την άσκηση 9.5 ώστε η ηλικία να υπολογίζεται μία φορά σε derived table. Εμφάνισε πλήθος πελατών ανά ομάδα για όλους τους πελάτες με γνωστή ημερομηνία γέννησης, ταξινομημένα κατά το κάτω όριο της ομάδας.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    label    | customers
-------------+-----------
 έως 30      |         6
 31 έως 45   |        13
 46 και πάνω |         7
(3 rows)
```

ΑΣΚΗΣΗ 14.5 Τα δύο ακριβότερα ανά κατηγορία

Για τις κατηγορίες 8, 9 και 10 εμφάνισε το όνομα της κατηγορίας και τα δύο ακριβότερα ενεργά προϊόντα της με την τιμή τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   category   |         name          |  price
--------------+-----------------------+---------
 Laptops      | Laptop Pro 16         | 2199.00
 Laptops      | Laptop Aero 14        | 1149.00
 Περιφερειακά | Οθόνη 27" 4K          |  329.00
 Περιφερειακά | USB-C docking station |  139.00
 Ήχος         | Πικάπ                 |  229.00
 Ήχος         | Ακουστικά ANC         |  199.00
(6 rows)
```

ΑΣΚΗΣΗ 14.6 Ο νεότερος πελάτης κάθε πόλης

Για κάθε πόλη εμφάνισε το όνομά της και τον πελάτη που εγγράφηκε πιο πρόσφατα με την ημερομηνία εγγραφής. Οι πόλεις χωρίς πελάτες πρέπει να εμφανίζονται με ∅. Ταξινόμησε κατά `id` πόλης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    city     | first_name |   joined
-------------+------------+------------
 Αθήνα       | Όλγα       | 2026-06-20
 Θεσσαλονίκη | Πέτρος     | 2026-01-15
 Πάτρα       | Θανάσης    | 2025-06-12
 Ηράκλειο    | Ηλίας      | 2026-04-03
 Λάρισα      | Μιχάλης    | 2025-08-28
 Βόλος       | Άννα       | 2025-09-26
 Ιωάννινα    | Ραλλού     | 2025-12-19
 Χανιά       | Λευτέρης   | 2025-11-27
 Καλαμάτα    | Ευαγγελία  | 2025-05-30
 Ρόδος       | Σπύρος     | 2025-07-19
 Κοζάνη      | ∅          | ∅
(11 rows)
```

ΑΣΚΗΣΗ 14.7 Η τελευταία κριτική

Για κάθε προϊόν με τουλάχιστον δύο κριτικές εμφάνισε όνομα, τη βαθμολογία και τον τίτλο της πιο πρόσφατης κριτικής του.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                name                 | rating |       title
-------------------------------------+--------+--------------------
 Ακουστικά ANC                       |      3 | Ok
 Ασύρματο ποντίκι                    |      1 | Χάλασε
 Βίος και πολιτεία του Αλέξη Ζορμπά |      5 | Ζορμπάς για πάντα
 Εργονομική καρέκλα                  |      4 | Άνετη
 Καφετιέρα espresso                  |      2 | Διαρροή
 Μαθαίνω Python                      |      4 | Πρακτικό
 Μηχανικό πληκτρολόγιο               |      4 | θορυβώδες
 Οθόνη 27" 4K                        |      3 | Μέτρια
 Laptop Aero 14                      |      2 | Ζεσταίνεται
 Laptop Pro 16                       |      3 | Ακριβό
 Laptop Student 15                   |      2 | Αργό
 PostgreSQL σε βάθος                 |      5 | Must have
(12 rows)
```

ΜΕΡΟΣ ΙΙΙ

Αλλαγές, σχήμα και transactions

Ως εδώ διαβάζαμε. Τώρα γράφουμε δεδομένα, ορίζουμε πίνακες με κανόνες που η βάση εγγυάται και ομαδοποιούμε αλλαγές σε transactions που είτε γίνονται ολόκληρες είτε καθόλου.

ΚΕΦΑΛΑΙΟ 15

INSERT, UPDATE, DELETE και RETURNING

Τρεις εντολές αλλάζουν δεδομένα. Το `INSERT` προσθέτει γραμμές, το `UPDATE` αλλάζει τιμές σε υπάρχουσες και το `DELETE` τις αφαιρεί. Το `UPDATE` και το `DELETE` χρησιμοποιούν το ίδιο `WHERE` που ήδη ξέρεις. Το `RETURNING` σου δείχνει τι άλλαξε.

> **ΣΗΜΕΙΩΣΗ**
> Τα παραδείγματα του κεφαλαίου αλλάζουν τη βάση. Όταν τελειώσεις, τρέξε ξανά το `bash data/setup.sh` για να επιστρέψεις στα αρχικά δεδομένα, αλλιώς τα αποτελέσματα των επόμενων κεφαλαίων θα διαφέρουν από του βιβλίου.

INSERT

Το `INSERT INTO πίνακας (στήλες) VALUES (τιμές)` προσθέτει μία γραμμή. Γράφε πάντα τη λίστα των στηλών. Χωρίς αυτήν οι τιμές αντιστοιχίζονται με τη σειρά των στηλών του πίνακα. Μια μελλοντική αλλαγή στον πίνακα θα τις μπερδέψει.

SQL

```
INSERT INTO cities (id, name, region, population)
VALUES (12, 'Τρίπολη', 'Πελοπόννησος', 30912);
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1
```

Το `psql` απάντησε `INSERT 0 1`. Ο δεύτερος αριθμός είναι οι γραμμές που μπήκαν. Ο πρώτος είναι κατάλοιπο από παλαιότερες εκδόσεις και είναι πάντα 0.

Όσες στήλες δεν αναφέρεις παίρνουν την τιμή `DEFAULT` του πίνακα ή `NULL` αν δεν έχουν. Στον `customers` το `id` είναι **identity column**. Η βάση το γεμίζει με τον επόμενο αριθμό μιας ακολουθίας. Το `newsletter` έχει default `false`.

SQL

```
INSERT INTO customers (first_name, last_name, email, city_id, created_at)
VALUES ('Νεφέλη', 'Αργυρίου', 'nefeli.a@example.gr', 12, '2026-06-29 10:00')
RETURNING id, newsletter;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | newsletter
----+------------
 31 | f
(1 row)

INSERT 0 1
```

Το `RETURNING` επιστρέφει στήλες από τις γραμμές που μόλις γράφτηκαν. Χωρίς αυτό θα έπρεπε να κάνεις δεύτερο query για να μάθεις το `id` που δόθηκε. Σε ένα σύστημα με πολλούς ταυτόχρονους χρήστες δεν θα ήξερες ποιο είναι το δικό σου.

Ένα `INSERT` μπορεί να περιέχει πολλές γραμμές, χωρισμένες με κόμμα. Είναι πολύ γρηγορότερο από πολλά ξεχωριστά `INSERT`.

SQL

```
INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (162, 14, 1, 24.90),
       (162, 23, 2, 34.90)
RETURNING *;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | product_id | quantity | unit_price
----------+------------+----------+------------
      162 |         14 |        1 |      24.90
      162 |         23 |        2 |      34.90
(2 rows)

INSERT 0 2
```

INSERT από SELECT

Αντί για `VALUES` μπορείς να γράψεις ένα `SELECT`. Κάθε γραμμή του αποτελέσματος γίνεται νέα γραμμή του πίνακα. Ας πούμε ότι η `Θεσσαλική Χαρτική` αρχίζει να προμηθεύει όλα τα βιβλία λογοτεχνίας με 5% πάνω από το κόστος τους.

SQL

```
INSERT INTO product_suppliers (product_id, supplier_id, supply_price)
SELECT p.id, 6, round(p.cost * 1.05, 2)
FROM products AS p
WHERE p.category_id = 2
  AND NOT EXISTS (SELECT 1 FROM product_suppliers AS ps
                  WHERE ps.product_id = p.id AND ps.supplier_id = 6)
RETURNING product_id, supply_price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 product_id | supply_price
------------+--------------
          1 |         9.56
          2 |         8.19
          4 |         8.40
(3 rows)

INSERT 0 3
```

Το `NOT EXISTS` αφήνει έξω το βιβλίο που η Θεσσαλική Χαρτική ήδη προμηθεύει. Χωρίς αυτό, το `INSERT` θα παραβίαζε το primary key του `product_suppliers`.

Όταν η βάση λέει όχι

Ο πίνακας έχει κανόνες και η βάση τους ελέγχει σε κάθε αλλαγή. Αν μια γραμμή παραβιάζει κάποιον, ολόκληρη η εντολή αποτυγχάνει και δεν γράφεται τίποτα.

SQL

```
INSERT INTO customers (first_name, last_name, city_id, created_at)
VALUES ('Άγνωστος', 'Πελάτης', 99, '2026-06-29');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  insert or update on table "customers" violates foreign key constraint "customers_city_id_fkey"
DETAIL:  Key (city_id)=(99) is not present in table "cities".
```

SQL

```
INSERT INTO products (sku, name, price)
VALUES ('XX-001', 'Δωρεάν δείγμα', 0);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  new row for relation "products" violates check constraint "products_price_check"
DETAIL:  Failing row contains (31, XX-001, Δωρεάν δείγμα, null, 0.00, null, 0, t, {}, {}, null).
```

Το πρώτο παραβιάζει το foreign key, αφού δεν υπάρχει πόλη 99. Το δεύτερο το `CHECK (price > 0)`. Το κεφάλαιο 17 δείχνει πώς ορίζεις τέτοιους κανόνες.

UPDATE

Το `UPDATE πίνακας SET στήλη = έκφραση WHERE συνθήκη` αλλάζει τις γραμμές που περνούν το `WHERE`. Η έκφραση μπορεί να χρησιμοποιεί την παλιά τιμή.

```sql
UPDATE products
SET price = round(price * 1.10, 2)
WHERE category_id = 13
RETURNING name, price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name             | price
-----------------------------+--------
 Εργονομική καρέκλα          | 317.90
 Γραφείο ρυθμιζόμενου ύψους | 504.90
 Φωτιστικό γραφείου LED      |  46.20
(3 rows)

UPDATE 3
```

Από την PostgreSQL 18 το `RETURNING` μπορεί να δείξει και την τιμή πριν από την αλλαγή, με τα προθέματα `old` και `new`.

```sql
UPDATE products
SET stock = stock - 1
WHERE id = 25
RETURNING name, old.stock AS before, new.stock AS after;
```

ΑΠΟΤΕΛΕΣΜΑ

```
        name         | before | after
---------------------+--------+-------
 Εργονομική καρέκλα |      6 |     5
(1 row)

UPDATE 1
```

ΠΑΓΙΔΑ

Ένα `UPDATE` χωρίς `WHERE` αλλάζει κάθε γραμμή του πίνακα. Πριν από ένα `UPDATE` ή `DELETE` σε δεδομένα που μετράνε, γράψε πρώτα ένα `SELECT` με το ίδιο `WHERE` και δες ποιες γραμμές επιστρέφει. Ακόμη καλύτερα, τρέξε το μέσα σε transaction που μπορείς να αναιρέσεις. Το κεφάλαιο 19 δείχνει πώς.

UPDATE με άλλους πίνακες

Όταν η νέα τιμή ή η συνθήκη εξαρτάται από άλλον πίνακα, η PostgreSQL έχει το `UPDATE ... FROM`. Ο πίνακας του `FROM` ενώνεται με τον πίνακα που αλλάζει μέσω του `WHERE`, όπως σε ένα join.

```sql
UPDATE products AS p
SET stock = p.stock + 5
FROM product_suppliers AS ps
WHERE ps.product_id = p.id
  AND ps.supplier_id = 3
  AND p.stock < 10
RETURNING p.name, p.stock;
```

ΑΠΟΤΕΛΕΣΜΑ

```
   name    | stock
-----------+-------
 Webcam HD |     5
 Πικάπ     |     8
(2 rows)

UPDATE 2
```

Ο προμηθευτής 3 έστειλε πέντε τεμάχια από κάθε προϊόν του που είχε απόθεμα κάτω από 10. Αν μια γραμμή του `products` ταίριαζε με πολλές γραμμές του `FROM`, θα άλλαζε μόνο μία φορά, με τιμή από μία τυχαία από αυτές. Όταν αυτό είναι πιθανό, γράψε subquery που δίνει μία τιμή ανά γραμμή.

DELETE

Το `DELETE FROM πίνακας WHERE συνθήκη` αφαιρεί γραμμές. Όπως το `UPDATE`, χωρίς `WHERE` αφαιρεί τα πάντα.

```sql
DELETE FROM reviews
WHERE rating = 1
RETURNING product_id, title;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 product_id | title
------------+--------
         14 | Χάλασε
(1 row)

DELETE 1
```

Τα foreign keys προστατεύουν και από διαγραφές. Δεν μπορείς να διαγράψεις μια πόλη στην οποία μένουν πελάτες, γιατί θα έμεναν πελάτες που δείχνουν σε πόλη που δεν υπάρχει.

```sql
DELETE FROM cities WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  update or delete on table "cities" violates foreign key constraint
"customers_city_id_fkey" on table "customers"
DETAIL:  Key (id)=(1) is still referenced from table "customers".
```

Το ανάλογο του `UPDATE ... FROM` είναι το `DELETE ... USING`. Εδώ διαγράφονται τα events των πελατών που εγγράφηκαν τον Ιούνιο του 2026.

SQL

```
DELETE FROM events AS e
USING customers AS c
WHERE c.id = e.customer_id
  AND c.created_at >= '2026-06-01'
RETURNING e.customer_id, e.kind;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | kind
-------------+------
          30 | view
          30 | view
          30 | view
          30 | view
(4 rows)

DELETE 4
```

Για να αδειάσεις ολόκληρο πίνακα υπάρχει το `TRUNCATE`. Είναι πολύ γρηγορότερο από το `DELETE` χωρίς `WHERE`, γιατί δεν εξετάζει γραμμή προς γραμμή, αλλά δεν επιστρέφει τι διέγραψε και κλειδώνει ολόκληρο τον πίνακα.

ΚΡΑΤΑ ΑΥΤΟ

Το `INSERT` προσθέτει γραμμές από `VALUES` ή από `SELECT`. Το `UPDATE` και το `DELETE` αλλάζουν μόνο ό,τι περνά το `WHERE` και μπορούν να δουν άλλους πίνακες με `FROM` και `USING`. Το `RETURNING` επιστρέφει τις γραμμές που άλλαξαν, με `old` και `new` από την PostgreSQL 18. Η βάση απορρίπτει κάθε αλλαγή που παραβιάζει constraint.

Ασκήσεις

Οι λύσεις εκτελέστηκαν μέσα σε transaction που αναιρέθηκε, οπότε κάθε άσκηση ξεκινά από την κατάσταση της βάσης στο τέλος της θεωρίας.

ΑΣΚΗΣΗ 15.1 Νέα πόλη

Πρόσθεσε την πόλη Κέρκυρα με `id` 13, περιφέρεια `Ιόνια Νησιά` και πληθυσμό 32095. Επέστρεψε ολόκληρη τη γραμμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |  name   |   region    | population
----+---------+-------------+------------
 13 | Κέρκυρα | Ιόνια Νησιά |      32095
(1 row)

INSERT 0 1
```

ΑΣΚΗΣΗ 15.2 Νέος πελάτης με newsletter

Πρόσθεσε πελάτη με όνομα `Ορέστης`, επώνυμο `Δανιήλ`, email `orestis.d@example.gr`, πόλη 7, εγγραφή στο newsletter και ημερομηνία εγγραφής `2026-06-30 09:00`. Επέστρεψε το `id` και το `created_at`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |       created_at
----+------------------------
 33 | 2026-06-30 09:00:00+03
(1 row)

INSERT 0 1
```

ΑΣΚΗΣΗ 15.3 Νέο προϊόν με κατηγορία από όνομα

Πρόσθεσε το προϊόν με `sku` `AU-005`, όνομα `Ακουστικά gaming`, τιμή 89, κόστος 50 και απόθεμα 12, στην κατηγορία με όνομα `Ήχος`. Βρες το `category_id` με `INSERT ... SELECT` και όχι γράφοντας τον αριθμό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |  sku   | category_id
----+--------+-------------
 32 | AU-005 |          10
(1 row)

INSERT 0 1
```

ΑΣΚΗΣΗ 15.4 Εκπτώσεις στην κουζίνα

Μείωσε κατά 15% τις τιμές των προϊόντων της κατηγορίας Κουζίνα, στρογγυλεμένες σε δύο δεκαδικά. Επέστρεψε το όνομα, την παλιά και τη νέα τιμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
        name        | old_price | new_price
--------------------+-----------+-----------
 Καφετιέρα espresso |    249.00 |    211.65
 Βραστήρας          |     34.90 |     29.67
 Σετ μαχαιριών      |     64.00 |     54.40
(3 rows)

UPDATE 3
```

ΑΣΚΗΣΗ 15.5 Παράδοση των παλιών αποστολών

Άλλαξε σε `delivered` την κατάσταση των παραγγελιών που είναι `shipped` και στάλθηκαν πριν από τις 20 Ιουνίου 2026. Επέστρεψε `id` και `shipped_at`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  |       shipped_at
-----+------------------------
 140 | 2026-06-11 10:09:00+03
 142 | 2026-06-17 18:32:00+03
 143 | 2026-06-16 21:35:00+03
 144 | 2026-06-18 11:46:00+03
 145 | 2026-06-15 22:04:00+03
 147 | 2026-06-16 02:41:00+03
 148 | 2026-06-18 12:47:00+03
 149 | 2026-06-19 16:05:00+03
 150 | 2026-06-17 00:06:00+03
(9 rows)

UPDATE 9
```

ΑΣΚΗΣΗ 15.6 Ανανέωση αποθέματος από παραγγελία

Η παραγγελία 157 ακυρώνεται. Αύξησε το απόθεμα κάθε προϊόντος της κατά την ποσότητα που είχε παραγγελθεί. Επέστρεψε όνομα και νέο απόθεμα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      name       | stock
-----------------+-------
 Ματωμένα χώματα |     1
 Σετ μαχαιριών   |    17
(2 rows)

UPDATE 2
```

ΑΣΚΗΣΗ 15.7 Διαγραφή πελάτη

Διέγραψε τον πελάτη 30. Πρώτα πρέπει να φύγουν τα events του, αλλιώς το foreign key θα εμποδίσει τη διαγραφή. Γράψε τις δύο εντολές με τη σωστή σειρά και επέστρεψε από τη δεύτερη το όνομα του πελάτη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
DELETE 0

 first_name | last_name
------------+-----------
 Όλγα       | Βλάχου
(1 row)

DELETE 1
```

ΑΣΚΗΣΗ 15.8 Αντίγραφο αρχείου

Διέγραψε τις κριτικές με βαθμολογία 2 που γράφτηκαν πριν από το 2026 και επέστρεψε όλες τις στήλες τους, ώστε να μπορείς να τις κρατήσεις κάπου αλλού.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | product_id | customer_id | rating |    title    |       created_at
----+------------+-------------+--------+-------------+------------------------
 11 |         22 |           8 |      2 | Διαρροή     | 2025-03-27 22:15:00+02
 30 |         13 |          13 |      2 | Αργό        | 2025-09-13 09:50:00+03
 36 |         11 |           9 |      2 | Ζεσταίνεται | 2025-11-01 14:33:00+02
(3 rows)

DELETE 3
```

ΚΕΦΑΛΑΙΟ 16

Upsert με ON CONFLICT και MERGE

Συχνά δεν ξέρεις αν μια γραμμή υπάρχει ήδη. Θέλεις να την προσθέσεις αν λείπει και να την ενημερώσεις αν υπάρχει. Το `INSERT ... ON CONFLICT` το κάνει σε μία ατομική εντολή. Το `MERGE` συγχρονίζει έναν πίνακα με μια πηγή και μπορεί επίσης να διαγράψει.

Το πρόβλημα

Ένας προμηθευτής στέλνει κάθε πρωί τις τιμές του. Για κάθε προϊόν πρέπει να ενημερώσεις την τιμή αν το προϊόν υπάρχει ήδη στον `product_suppliers` ή να το προσθέσεις αν είναι καινούριο. Η πρώτη ιδέα είναι να ελέγξεις με `SELECT` και μετά να αποφασίσεις. Σε σύστημα με έναν χρήστη δουλεύει. Όταν δύο διεργασίες κάνουν τον ίδιο έλεγχο την ίδια στιγμή, βλέπουν και οι δύο ότι η γραμμή λείπει. Κάνουν και οι δύο `INSERT` και η δεύτερη αποτυγχάνει. Το κεφάλαιο 38 δείχνει αυτό το παράθυρο αναλυτικά.

Η λύση είναι να αφήσεις τη βάση να αποφασίσει, με βάση ένα unique constraint που ήδη εγγυάται ότι δεν υπάρχουν διπλές γραμμές.

ON CONFLICT DO NOTHING

Η πιο απλή μορφή αγνοεί τη γραμμή όταν συγκρούεται με unique constraint.

SQL

```
INSERT INTO cities (id, name, region, population)
VALUES (11, 'Κοζάνη', 'Δυτική Μακεδονία', 71388)
ON CONFLICT DO NOTHING;
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 0
```

Η απάντηση `INSERT 0 0` λέει ότι δεν μπήκε τίποτα και δεν υπήρξε σφάλμα. Σε ένα `INSERT` πολλών γραμμών οι υπόλοιπες μπαίνουν κανονικά. Χρήσιμο όταν φορτώνεις δεδομένα που ίσως έχεις ήδη.

ON CONFLICT DO UPDATE

Για να ενημερώσεις την υπάρχουσα γραμμή πρέπει να πεις σε ποιο constraint αφορά η σύγκρουση. Γράφεις τις στήλες του μέσα σε παρενθέσεις και η PostgreSQL βρίσκει το unique index που τους αντιστοιχεί. Μέσα στο `DO UPDATE` το ειδικό όνομα `excluded` είναι η γραμμή που προσπάθησες να βάλεις.

SQL

```
INSERT INTO product_suppliers (product_id, supplier_id, supply_price)
VALUES (19, 2, 118.00),
       (20, 2, 44.00)
ON CONFLICT (product_id, supplier_id)
DO UPDATE SET supply_price = excluded.supply_price
RETURNING product_id, supplier_id, old.supply_price AS old_price,
          new.supply_price AS new_price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 product_id | supplier_id | old_price | new_price
------------+-------------+-----------+-----------
         19 |           2 |    124.80 |    118.00
         20 |           2 |         ø |     44.00
(2 rows)

INSERT 0 2
```

Το προϊόν 19 υπήρχε για τον προμηθευτή 2 και ενημερώθηκε. Το 20 δεν υπήρχε και μπήκε. Η διάκριση φαίνεται στη στήλη `old_price`. Όταν η γραμμή είναι καινούρια δεν υπάρχει παλιά τιμή, οπότε το `old` είναι `NULL`. Αυτό το `RETURNING` με `old` και `new` υπάρχει από την PostgreSQL 18.

ΠΑΓΙΔΑ

Η στήλη του `ON CONFLICT (...)` πρέπει να έχει unique constraint ή unique index. Αν δεν έχει, η εντολή αποτυγχάνει. Το upsert δεν ψάχνει γραμμή με ίδια τιμή. Βασίζεται στην εγγύηση μοναδικότητας για να αποφασίσει ατομικά.

SQL

```
INSERT INTO customers (first_name, last_name, phone, created_at)
VALUES ('Μαρία', 'Παπαδοπούλου', '6944123456', '2026-06-30')
ON CONFLICT (phone) DO NOTHING;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  there is no unique or exclusion constraint matching the ON CONFLICT specification
```

Update υπό συνθήκη

Το `DO UPDATE` δέχεται δικό του `WHERE`. Αν η συνθήκη δεν ισχύει, η γραμμή μένει όπως είναι και το `INSERT` δεν μετράει την ενημέρωση. Ο προμηθευτής στέλνει τιμές, αλλά εσύ θέλεις να κρατάς μόνο τις μειώσεις.

```sql
INSERT INTO product_suppliers (product_id, supplier_id, supply_price)
VALUES (11, 2, 905.00),
       (12, 2, 1690.00)
ON CONFLICT (product_id, supplier_id)
DO UPDATE SET supply_price = excluded.supply_price
WHERE excluded.supply_price < product_suppliers.supply_price
RETURNING product_id, new.supply_price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 product_id | supply_price
------------+--------------
         12 |      1690.00
(1 row)

INSERT 0 1
```

Μόνο το προϊόν 12 άλλαξε, γιατί το 11 είχε ήδη χαμηλότερη τιμή. Μέσα στο `WHERE` ο πίνακας στόχος αναφέρεται με το όνομά του, εδώ `product_suppliers`. Η νέα γραμμή αναφέρεται ως `excluded`.

MERGE

Το `MERGE` συγκρίνει γραμμή προς γραμμή έναν πίνακα στόχο με μια πηγή και εκτελεί διαφορετική ενέργεια ανάλογα με το αν βρέθηκε ταίρι. Η πηγή μπορεί να είναι πίνακας, subquery ή λίστα `VALUES`. Ας πούμε ότι ο προμηθευτής 2 δεν στέλνει μόνο αλλαγές αλλά ολόκληρη τη λίστα των προϊόντων που προμηθεύει. Ό,τι λείπει από τη λίστα δεν το προμηθεύει πια.

```sql
MERGE INTO product_suppliers AS t
USING (VALUES (11, 880.00), (12, 1690.00), (13, 515.00),
              (14, 11.00), (15, 47.00), (17, 85.00),
              (21, 66.00)) AS s(product_id, supply_price)
ON t.product_id = s.product_id AND t.supplier_id = 2
WHEN MATCHED AND t.supply_price <> s.supply_price THEN
  UPDATE SET supply_price = s.supply_price
WHEN NOT MATCHED THEN
  INSERT (product_id, supplier_id, supply_price)
  VALUES (s.product_id, 2, s.supply_price)
WHEN NOT MATCHED BY SOURCE AND t.supplier_id = 2 THEN
  DELETE
RETURNING merge_action() AS action, t.product_id,
          old.supply_price AS old_price, new.supply_price AS new_price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 action | product_id | old_price | new_price
--------+------------+-----------+-----------
 UPDATE |         11 |    890.00 |    880.00
 UPDATE |         13 |    520.00 |    515.00
 UPDATE |         15 |     48.00 |     47.00
 DELETE |         16 |    250.00 |         ∅
 DELETE |         18 |     30.00 |         ∅
 DELETE |         29 |    610.00 |         ∅
 DELETE |         19 |    118.00 |         ∅
 DELETE |         20 |     44.00 |         ∅
 INSERT |         21 |         ∅ |     66.00
(9 rows)

MERGE 9
```

Τρεις περιπτώσεις καλύπτονται σε μία εντολή. Το `WHEN MATCHED` αφορά γραμμές που υπάρχουν και στις δύο πλευρές. Εδώ έχει και επιπλέον συνθήκη, ώστε να μην ενημερώνονται γραμμές χωρίς αλλαγή. Το `WHEN NOT MATCHED` αφορά γραμμές της πηγής χωρίς ταίρι στον στόχο. Το `WHEN NOT MATCHED BY SOURCE` αφορά γραμμές του στόχου χωρίς ταίρι στην πηγή. Η συνθήκη `t.supplier_id = 2` είναι κρίσιμη. Χωρίς αυτήν θα διαγράφονταν και όλες οι γραμμές των άλλων προμηθευτών.

Η συνάρτηση `merge_action()` στο `RETURNING` λέει ποια ενέργεια έγινε σε κάθε γραμμή. Το `RETURNING` στο `MERGE` και το `WHEN NOT MATCHED BY SOURCE` υπάρχουν από την PostgreSQL 17.

Πότε ποιο

Το `INSERT ... ON CONFLICT` είναι ασφαλές όταν πολλές διεργασίες γράφουν ταυτόχρονα. Η βάση χρησιμοποιεί το unique index για να αποφασίσει ατομικά και δεν θα δεις ποτέ σφάλμα μοναδικότητας από δύο ταυτόχρονα upserts. Είναι η σωστή επιλογή για upsert γραμμή προς γραμμή σε εφαρμογή.

Το `MERGE` είναι πιο εκφραστικό. Κάνει και διαγραφές, δέχεται πολλές συνθήκες και δεν χρειάζεται unique constraint. Δεν έχει όμως την ίδια εγγύηση. Αν δύο `MERGE` τρέξουν ταυτόχρονα και αποφασίσουν και τα δύο `INSERT` για την ίδια γραμμή, το δεύτερο θα αποτύχει με σφάλμα μοναδικότητας. Είναι η σωστή επιλογή για συγχρονισμό από μια πηγή, συνήθως σε εργασίες που τρέχουν μία τη φορά.

> **ΚΡΑΤΑ ΑΥΤΟ**
>
> Το `ON CONFLICT DO NOTHING` αγνοεί συγκρούσεις και το `ON CONFLICT (...) DO UPDATE` ενημερώνει με τις τιμές του `excluded`. Και τα δύο στηρίζονται σε unique constraint και είναι ατομικά. Το `MERGE` καλύπτει ενημέρωση, εισαγωγή και διαγραφή σε μία εντολή για συγχρονισμό από πηγή.

Ασκήσεις

ΑΣΚΗΣΗ 16.1 Εισαγωγή χωρίς διπλά

Πρόσθεσε τις πόλεις Χανιά (8) και Άρτα (14, Ήπειρος, 43166) με ένα `INSERT`, αγνοώντας όσες υπάρχουν ήδη. Επέστρεψε τις γραμμές που μπήκαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | name | region  | population
----+------+---------+------------
 14 | Άρτα | Ήπειρος |      43166
(1 row)

INSERT 0 1
```

ΑΣΚΗΣΗ 16.2 Ενημέρωση επαφής από εισαγωγή

Ένα αρχείο φέρνει δύο πελάτες με email. Ο `maria.p@example.gr` υπάρχει ήδη και πρέπει να πάρει νέο τηλέφωνο `6900000001`. Ο `lina.k@example.gr` είναι νέος, με όνομα `Λίνα Καλλέργη` και τηλέφωνο `6900000002`. Χρησιμοποίησε ημερομηνία εγγραφής `2026-06-30`. Επέστρεψε `id`, email, αν η γραμμή είναι νέα και το νέο τηλέφωνο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |       email        | inserted |   phone
----+--------------------+----------+------------
  1 | maria.p@example.gr | f        | 6900000001
 32 | lina.k@example.gr  | t        | 6900000002
(2 rows)

INSERT 0 2
```

ΑΣΚΗΣΗ 16.3 Μόνο ανατιμήσεις

Ο προμηθευτής 1 στέλνει νέες τιμές για τα προϊόντα 1, 2 και 3, 9.50, 7.50 και 4.60. Κάνε upsert αλλά ενημέρωσε μια υπάρχουσα τιμή μόνο αν η νέα είναι μεγαλύτερη. Επέστρεψε `product_id` και νέα τιμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 product_id | supply_price
------------+--------------
          1 |         9.50
          3 |         4.60
(2 rows)

INSERT 0 2
```

ΑΣΚΗΣΗ 16.4 Συγχρονισμός τιμοκαταλόγου

Με `MERGE`, συγχρόνισε τον `products` με τη λίστα `('PR-001', 22.90)`, `('PR-002', 89.00)` και `('PR-006', 19.90)`. Για υπάρχοντα `sku` ενημέρωσε την τιμή μόνο αν διαφέρει. Για νέα `sku` πρόσθεσε προϊόν με όνομα `Mousepad XL`, κατηγορία 9 και απόθεμα 50. Επέστρεψε ενέργεια, `sku` και τιμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 action |  sku   | price
--------+--------+-------
 UPDATE | PR-001 | 22.90
 INSERT | PR-006 | 19.90
(2 rows)

MERGE 2
```

ΑΣΚΗΣΗ 16.5 Πλήρης λίστα προμηθευτή

Ο προμηθευτής 3 στέλνει την πλήρη λίστα του, `(16, 245.00)`, `(19, 118.00)` και `(20, 41.00)`. Με `MERGE`, ενημέρωσε τις υπάρχουσες τιμές, πρόσθεσε τα καινούρια και διέγραψε τα προϊόντα του προμηθευτή 3 που δεν είναι στη λίστα. Επέστρεψε ενέργεια και `product_id`, ταξινομημένα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 action | product_id
--------+------------
 UPDATE |         16
 DELETE |         18
 UPDATE |         19
 UPDATE |         20
 DELETE |         21
 DELETE |         30
(6 rows)

MERGE 6
```

ΑΣΚΗΣΗ 16.6 Upsert χωρίς unique

Δοκίμασε να κάνεις upsert στον `reviews` με `ON CONFLICT (product_id, customer_id) DO NOTHING`, για μια δεύτερη κριτική του πελάτη 1 στο προϊόν 19. Εξήγησε το σφάλμα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  there is no unique or exclusion constraint matching the ON CONFLICT specification
```

ΚΕΦΑΛΑΙΟ 17

CREATE TABLE, τύποι και constraints

Ένας πίνακας δεν είναι μόνο στήλες. Είναι και κανόνες που η βάση ελέγχει σε κάθε αλλαγή. Κάθε κανόνας που γράφεις ως constraint είναι ένας έλεγχος που δεν χρειάζεται να θυμάται κανένα κομμάτι της εφαρμογής.

CREATE TABLE

Το κατάστημα θέλει να κρατά κουπόνια με ποσοστό έκπτωσης και περίοδο ισχύος. Ο πίνακας ορίζεται με όνομα και λίστα στηλών, καθεμία με τύπο και προαιρετικούς κανόνες.

SQL

```
CREATE TABLE coupons (
  code         text PRIMARY KEY,
  discount_pct numeric(5,2) NOT NULL CHECK (discount_pct > 0 AND discount_pct <= 100),
  valid_from   date NOT NULL,
  valid_until  date,
  max_uses     integer CHECK (max_uses > 0),
  active       boolean NOT NULL DEFAULT true,
  CONSTRAINT coupons_period CHECK (valid_until > valid_from)
);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE
```

Οι κανόνες που αφορούν μία στήλη γράφονται δίπλα της. Όσοι αφορούν πολλές στήλες γράφονται ως χωριστό στοιχείο της λίστας, όπως το `coupons_period`. Το `CONSTRAINT όνομα` δίνει όνομα στον κανόνα. Χωρίς αυτό η PostgreSQL φτιάχνει ένα όνομα μόνη της, όπως το `products_price_check` που είδες στο κεφάλαιο 15. Ένα καλό όνομα κάνει το μήνυμα λάθους κατανοητό.

SQL

```
INSERT INTO coupons (code, discount_pct, valid_from, valid_until)
VALUES ('WELCOME10', 10, '2025-01-01', NULL),
       ('SUMMER25', 25, '2025-06-01', '2025-09-01'),
       ('BLACKFRIDAY', 30, '2025-11-20', '2025-12-01');
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 3
```

```sql
INSERT INTO coupons (code, discount_pct, valid_from, valid_until)
VALUES ('WINTER', 15, '2026-02-01', '2026-01-01');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  new row for relation "coupons" violates check constraint "coupons_period"
DETAIL:  Failing row contains (WINTER, 15.00, 2026-02-01, 2026-01-01, null, t).
```

Το `WELCOME10` δεν έχει ημερομηνία λήξης. Το `CHECK (valid_until > valid_from)` δίνει `NULL` για αυτό και ένα `CHECK` απορρίπτει μόνο όταν η συνθήκη είναι ψευδής, όχι όταν είναι άγνωστη. Αυτή είναι η αντίθετη συμπεριφορά από το `WHERE`. Αν θέλεις να απαγορεύσεις και το `NULL`, πρόσθεσε `NOT NULL`.

Επιλογή τύπων

Για τις περισσότερες στήλες η επιλογή είναι απλή αν ακολουθήσεις λίγους κανόνες.

Για	Τύπος	Σημείωση
κείμενο	`text`	Το `varchar(n)` δεν είναι γρηγορότερο. Αν θέλεις όριο, γράψε `CHECK (length(x) <= n)`.
χρήματα	`numeric(p,s)`	Ποτέ `real` ή `double precision`.
μετρήσεις και κλειδιά	`integer` ή `bigint`	`bigint` για πίνακες που μπορεί να ξεπεράσουν τα δύο δισεκατομμύρια γραμμές.
στιγμές	`timestamptz`	Ο `timestamp` χωρίς ζώνη μόνο για ώρες τοίχου.
ημέρες	`date`	
ναι ή όχι	`boolean`	Όχι `integer` με 0 και 1.
ημιδομημένα δεδομένα	`jsonb`	Κεφάλαιο 27.
τυχαία αναγνωριστικά	`uuid`	Η `uuidv7()` της PostgreSQL 18 παράγει χρονικά ταξινομημένα uuid.

Identity columns

Για αυτόματους αριθμούς χρησιμοποιείς identity column. Υπάρχουν δύο μορφές. Με `GENERATED ALWAYS AS IDENTITY` η βάση αρνείται να δεχτεί τιμή από εσένα, εκτός αν γράψεις ρητά `OVERRIDING SYSTEM VALUE`. Με `GENERATED BY DEFAULT AS IDENTITY`, που χρησιμοποιεί το σχήμα του βιβλίου, δίνει αριθμό μόνο όταν δεν δώσεις εσύ. Η πρώτη μορφή προστατεύει από λάθη και είναι η καλύτερη προεπιλογή.

Σε παλαιότερο κώδικα θα δεις τον τύπο `serial`. Κάνει περίπου το ίδιο με παλαιότερο μηχανισμό. Σε νέους πίνακες γράψε identity.

Foreign keys και διαγραφές

Μια λίστα επιθυμιών συνδέει πελάτες και προϊόντα σε σχέση πολλά προς πολλά. Το primary key είναι το ζεύγος, οπότε ένα προϊόν μπαίνει μία φορά στη λίστα κάθε πελάτη.

SQL

```
CREATE TABLE wishlist_items (
  customer_id integer NOT NULL REFERENCES customers (id) ON DELETE CASCADE,
  product_id  integer NOT NULL REFERENCES products (id),
  added_at    timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (customer_id, product_id)
);

INSERT INTO wishlist_items (customer_id, product_id)
VALUES (26, 12), (26, 19), (30, 22);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 3
```

Το `ON DELETE CASCADE` λέει τι γίνεται όταν διαγραφεί ο πελάτης. Οι γραμμές του στη λίστα διαγράφονται μαζί του. Χωρίς αυτό, η διαγραφή του πελάτη θα αποτύγχανε, όπως απέτυχε η διαγραφή της πόλης στο κεφάλαιο 15. Οι επιλογές είναι τέσσερις.

Επιλογή	Όταν διαγραφεί η γονική γραμμή
`NO ACTION`, η προεπιλογή	η διαγραφή αποτυγχάνει αν υπάρχουν παιδιά
`RESTRICT`	το ίδιο, με έλεγχο αμέσως και όχι στο τέλος της εντολής
`CASCADE`	διαγράφονται και τα παιδιά
`SET NULL`	τα παιδιά μένουν με `NULL` στη στήλη

ΠΑΓΙΔΑ

Το `CASCADE` είναι βολικό για δεδομένα που δεν έχουν νόημα χωρίς τον γονέα, όπως μια λίστα επιθυμιών. Για δεδομένα με αξία, όπως παραγγελίες και πληρωμές, είναι επικίνδυνο. Μια διαγραφή πελάτη θα έσβηνε σιωπηλά λογιστικό ιστορικό. Εκεί θέλεις η διαγραφή να αποτύχει.

ALTER TABLE

Οι πίνακες αλλάζουν. Το `ALTER TABLE` προσθέτει και αφαιρεί στήλες και constraints. Τα κουπόνια των παραγγελιών μπορούν τώρα να δείχνουν στον `coupons`.

SQL

```
ALTER TABLE orders
  ADD CONSTRAINT orders_coupon_fkey
  FOREIGN KEY (coupon_code) REFERENCES coupons (code);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE
```

Όταν προσθέτεις constraint σε πίνακα με δεδομένα, η βάση ελέγχει όλες τις υπάρχουσες γραμμές. Αν μία το παραβιάζει, το `ALTER TABLE` αποτυγχάνει. Ο κανόνας «μία κριτική ανά πελάτη και προϊόν» δεν μπορεί να προστεθεί, επειδή ο πελάτης 2 έχει ήδη γράψει δύο κριτικές για το ίδιο laptop.

```sql
ALTER TABLE reviews
  ADD CONSTRAINT reviews_one_per_customer UNIQUE (product_id, customer_id);
```

```
ERROR:  could not create unique index "reviews_one_per_customer"
DETAIL:  Key (product_id, customer_id)=(11, 2) is duplicated.
```

Πρώτα καθαρίζεις τα δεδομένα και μετά προσθέτεις τον κανόνα. Σε μεγάλους πίνακες υπάρχει και η επιλογή `NOT VALID`, που εφαρμόζει τον κανόνα μόνο στις νέες γραμμές. Αργότερα το `VALIDATE CONSTRAINT` ελέγχει τις παλιές χωρίς να μπλοκάρει τις εγγραφές.

Για στήλες το `ALTER TABLE` έχει τις δικές του μορφές.

```sql
ALTER TABLE products ADD COLUMN vat_rate numeric(4,2) NOT NULL DEFAULT 24;
ALTER TABLE products ALTER COLUMN description SET NOT NULL;
ALTER TABLE products RENAME COLUMN vat_rate TO vat_pct;
SELECT name, price, vat_pct FROM products WHERE id <= 3;
```

```
ALTER TABLE

ALTER TABLE

ALTER TABLE

                name                | price | vat_pct
------------------------------------+-------+---------
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90 |   24.00
 Το Κιβώτιο                         | 14.50 |   24.00
 Η Φόνισσα                          |  9.90 |   24.00
(3 rows)
```

Η νέα στήλη πήρε την τιμή 24 σε όλες τις υπάρχουσες γραμμές. Από την PostgreSQL 11 αυτό γίνεται αμέσως, χωρίς να ξαναγραφτεί ο πίνακας, όταν το default είναι σταθερά.

UNIQUE και NULL

Ένα `UNIQUE` constraint επιτρέπει πολλές γραμμές με `NULL`, αφού δύο άγνωστες τιμές δεν θεωρούνται ίσες. Ο πίνακας `customers` έχει δύο πελάτες χωρίς email και το `UNIQUE (email)` δεν ενοχλείται. Αν θέλεις να επιτρέπεται το πολύ ένα `NULL`, γράφεις `UNIQUE NULLS NOT DISTINCT`, διαθέσιμο από την PostgreSQL 15.

Ένα `UNIQUE` είναι επίσης ευαίσθητο στα κεφαλαία. Το email του πελάτη 21 έχει κεφαλαία γράμματα. Ένας νέος πελάτης με το ίδιο email σε πεζά θα περνούσε το `UNIQUE (email)`. Για μοναδικότητα χωρίς διάκριση πεζών και κεφαλαίων χρειάζεσαι unique index πάνω στην έκφραση `lower(email)`. Θα το δούμε στο κεφάλαιο 30.

Generated columns

Μια generated column υπολογίζεται από άλλες στήλες της ίδιας γραμμής. Δεν γράφεις τιμή σε αυτήν. Η βάση την υπολογίζει.

SQL

```
ALTER TABLE order_items
  ADD COLUMN line_total numeric GENERATED ALWAYS AS (quantity * unit_price);

SELECT order_id, product_id, quantity, unit_price, line_total
FROM order_items
WHERE order_id = 144;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

 order_id | product_id | quantity | unit_price | line_total
----------+------------+----------+------------+------------
      144 |         24 |        2 |      64.00 |     128.00
      144 |          4 |        2 |      15.20 |      30.40
      144 |         14 |        1 |      24.90 |      24.90
(3 rows)
```

Στην PostgreSQL 18 μια generated column είναι από προεπιλογή `VIRTUAL`. Δεν αποθηκεύεται και υπολογίζεται όταν τη διαβάζεις. Με `STORED` αποθηκεύεται και υπολογίζεται σε κάθε αλλαγή της γραμμής. Η πρώτη μορφή δεν πιάνει χώρο. Η δεύτερη διαβάζεται φθηνότερα και μπορεί να έχει index.

DROP TABLE

Το `DROP TABLE` σβήνει έναν πίνακα με όλα τα δεδομένα του. Αν άλλοι πίνακες έχουν foreign key προς αυτόν, αποτυγχάνει, εκτός αν γράψεις `CASCADE`, που σβήνει και τα foreign keys. Το `IF EXISTS` αποφεύγει το σφάλμα όταν ο πίνακας δεν υπάρχει.

SQL

```
DROP TABLE IF EXISTS wishlist_items;
```

ΑΠΟΤΕΛΕΣΜΑ

```
DROP TABLE
```

ΚΡΑΤΑ ΑΥΤΟ

Κάθε κανόνας των δεδομένων που μπορεί να γραφτεί ως `NOT NULL`, `CHECK`, `UNIQUE`, `PRIMARY KEY` ή `FOREIGN KEY` πρέπει να γραφτεί εκεί. Το `CHECK` απορρίπτει μόνο το ψευδές, όχι το άγνωστο. Το `ON DELETE` αποφασίζει τι γίνεται με τα παιδιά μιας γραμμής που διαγράφεται. Το `ALTER TABLE` ελέγχει τα υπάρχοντα δεδομένα πριν δεχτεί έναν νέο κανόνα.

Ασκήσεις

ΑΣΚΗΣΗ 17.1 Πίνακας καταστημάτων

Φτιάξε πίνακα `stores` με `id` identity που δεν δέχεται τιμή από τον χρήστη, `name` κείμενο υποχρεωτικό και μοναδικό, `city_id` που δείχνει στον `cities`, `opened_on` ημερομηνία υποχρεωτική και `is_open` boolean με προεπιλογή `true`. Πρόσθεσε το κατάστημα `Κέντρο Αθήνας` στην πόλη 1 με ημερομηνία έναρξης `2024-03-01` και επέστρεψε ολόκληρη τη γραμμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

 id |     name      | city_id | opened_on  | is_open
----+---------------+---------+------------+---------
  1 | Κέντρο Αθήνας |       1 | 2024-03-01 | t
(1 row)

INSERT 0 1
```

ΑΣΚΗΣΗ 17.2 Κόστος μικρότερο από τιμή

Πρόσθεσε στον `products` constraint με όνομα `products_cost_below_price` που απαιτεί το κόστος να μην ξεπερνά την τιμή. Μετά δοκίμασε να αλλάξεις το κόστος του προϊόντος 1 σε 20.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

ERROR:  new row for relation "products" violates check constraint "products_cost_below_price"
DETAIL:  Failing row contains (1, BK-LIT-001, Βίος και πολιτεία του Αλέξη Ζορμπά, 2, 16.90,
20.00, 42, t, {βιβλίο,λογοτεχνία,bestseller}, {"pages": 432, "language": "el", "publisher":
"Καζαντζά..., Το μυθιστόρημα του Νίκου Καζαντζάκ..., 24.00).
```

ΑΣΚΗΣΗ 17.3 Διπλές κριτικές

Βρες τα ζεύγη προϊόντος και πελάτη με περισσότερες από μία κριτική. Κράτα την πιο πρόσφατη κριτική κάθε ζεύγους, διέγραψε τις υπόλοιπες και πρόσθεσε το constraint `reviews_one_per_customer`. Επέστρεψε τα `id` που διαγράφηκαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | product_id | customer_id
----+------------+-------------
 25 |         11 |           2
(1 row)

DELETE 1

ALTER TABLE
```

ΑΣΚΗΣΗ 17.4 Newsletter με SET NULL

Φτιάξε πίνακα `newsletter_sends` με `id` identity, `customer_id` που δείχνει στον `customers` με `ON DELETE SET NULL` και `sent_at timestamptz` υποχρεωτικό. Βάλε δύο αποστολές στον πελάτη 26 στις `2026-06-01 09:00` και `2026-06-15 09:00`. Διέγραψε τον πελάτη 26 και εμφάνισε τον πίνακα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 2

DELETE 20

DELETE 1

 id | customer_id |        sent_at
----+-------------+------------------------
  1 |           ø | 2026-06-01 09:00:00+03
  2 |           ø | 2026-06-15 09:00:00+03
(2 rows)
```

ΑΣΚΗΣΗ 17.5 Υπολογισμένο περιθώριο

Πρόσθεσε στον `products` generated column `margin` ίση με `price - cost`. Εμφάνισε όνομα, τιμή και `margin` για τα τρία προϊόντα με το μεγαλύτερο περιθώριο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

       name        |  price  | margin
-------------------+---------+--------
 Laptop Pro 16     | 2199.00 | 479.00
 Laptop Aero 14    | 1149.00 | 259.00
 Laptop Classic 13 |  799.00 | 189.00
(3 rows)
```

ΑΣΚΗΣΗ 17.6 Κουπόνι μίας χρήσης ανά πελάτη

Φτιάξε πίνακα `coupon_redemptions` με `coupon_code` που δείχνει στον `coupons`, `customer_id` που δείχνει στον `customers`, `order_id` που δείχνει στον `orders` και είναι μοναδικό. Πρόσθεσε και κανόνα ότι ο ίδιος πελάτης δεν εξαργυρώνει το ίδιο κουπόνι δύο φορές. Βάλε μια εξαργύρωση του `WELCOME10` από τον πελάτη 1 στην παραγγελία 11 και δοκίμασε να βάλεις δεύτερη στην παραγγελία 17.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 1

ERROR:  duplicate key value violates unique constraint "coupon_redemptions_pkey"
DETAIL:  Key (coupon_code, customer_id)=(WELCOME10, 1) already exists.
```

ΚΕΦΑΛΑΙΟ 18

Σχεδίαση σχήματος και κανονικοποίηση

Ένα καλό σχήμα γράφει κάθε γεγονός σε ένα σημείο. Όταν το ίδιο γεγονός γράφεται σε πολλές γραμμές, αργά ή γρήγορα οι γραμμές θα διαφωνήσουν. Η κανονικοποίηση είναι ένα σύνολο κανόνων που εντοπίζει αυτές τις επαναλήψεις.

Ένας επίπεδος πίνακας

Φαντάσου ότι το κατάστημα ξεκίνησε με ένα λογιστικό φύλλο. Κάθε γραμμή ήταν μια γραμμή παραγγελίας με όλα τα στοιχεία δίπλα της. Το `CREATE TABLE ... AS SELECT` φτιάχνει έναν τέτοιο πίνακα από το αποτέλεσμα ενός query.

SQL

```
CREATE TABLE flat_orders AS
SELECT o.id AS order_id, o.created_at,
       c.first_name || ' ' || c.last_name AS customer_name, c.email,
       t.name AS city, t.region,
       p.name AS product, oi.quantity, oi.unit_price
FROM orders AS o
JOIN customers AS c ON c.id = o.customer_id
JOIN cities AS t ON t.id = c.city_id
JOIN order_items AS oi ON oi.order_id = o.id
JOIN products AS p ON p.id = oi.product_id;

SELECT order_id, customer_name, city, region, product
FROM flat_orders
WHERE customer_name = 'Κατερίνα Σταύρου'
ORDER BY order_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 303

 order_id |  customer_name   | city  | region |              product
----------+------------------+-------+--------+-----------------------------------
       20 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Η Φόνισσα
       20 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Δίκτυα υπολογιστών
       20 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Σετ μαχαιριών
       35 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Βίος και πολιτεία του Αλέξη Ζορμπά
       35 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Ασύρματο ποντίκι
       35 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Laptop Student 15
       40 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Δωροκάρτα 50€
       40 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Ruby από την αρχή
       40 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Καφετιέρα espresso
       40 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Laptop Pro 16
       41 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Εργονομική καρέκλα
       82 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | PostgreSQL σε βάθος
       82 | Κατερίνα Σταύρου | Χανιά | Κρήτη  | Μικρόφωνο USB
(13 rows)
```

Ο πίνακας απαντά κάθε ερώτηση χωρίς joins. Έχει όμως τρία προβλήματα που λέγονται **ανωμαλίες**.

Η πρώτη είναι η **ανωμαλία ενημέρωσης**. Το email ενός πελάτη είναι γραμμένο σε κάθε γραμμή κάθε παραγγελίας του. Όταν αλλάξει, πρέπει να αλλάξουν όλες οι γραμμές. Αν ξεχαστεί μία, η βάση έχει δύο email για τον ίδιο άνθρωπο. Το ίδιο ισχύει για την περιφέρεια μιας πόλης.

SQL

```
UPDATE flat_orders
SET region = 'Κρήτη και Νησιά'
WHERE city = 'Χανιά' AND order_id < 50;

SELECT city, region, count(*) AS rows
FROM flat_orders
WHERE city = 'Χανιά'
GROUP BY city, region;
```

ΑΠΟΤΕΛΕΣΜΑ

```
UPDATE 11

 city  |     region      | rows
-------+-----------------+------
 Χανιά | Κρήτη           |   15
 Χανιά | Κρήτη και Νησιά |   11
(2 rows)
```

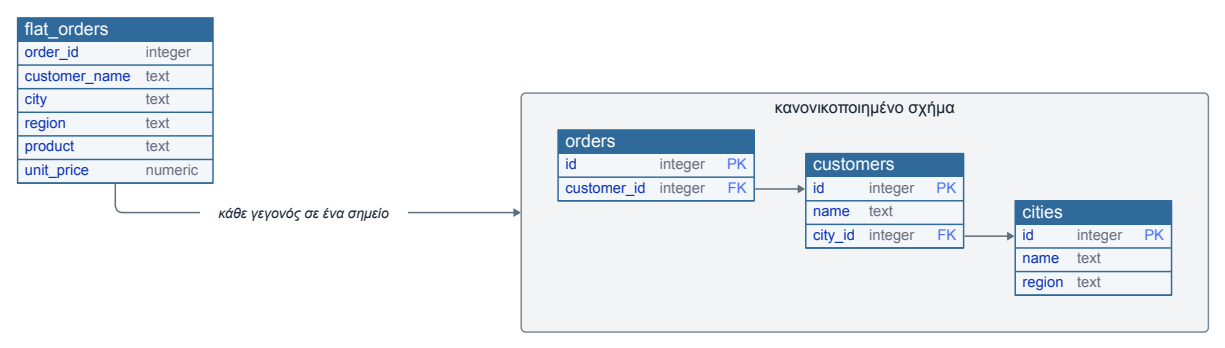
Τώρα τα Χανιά ανήκουν σε δύο περιφέρειες. Κανένα constraint δεν μπορεί να το εμποδίσει, αφού ο πίνακας δεν ξέρει ότι η περιφέρεια εξαρτάται από την πόλη.

Η δεύτερη είναι η **ανωμαλία εισαγωγής**. Δεν μπορείς να καταχωρίσεις έναν πελάτη πριν κάνει παραγγελία ή μια πόλη χωρίς πελάτη, αφού κάθε γραμμή είναι γραμμή παραγγελίας. Η τρίτη είναι η **ανωμαλία διαγραφής**. Αν διαγράψεις την τελευταία παραγγελία ενός πελάτη, χάνεις και τον πελάτη.

Εξαρτήσεις

Η ρίζα των προβλημάτων είναι οι **συναρτησιακές εξαρτήσεις**. Λέμε ότι το B εξαρτάται από το A όταν κάθε τιμή του A αντιστοιχεί σε μία μόνο τιμή του B. Η περιφέρεια εξαρτάται από την πόλη. Το email εξαρτάται από τον πελάτη. Η τιμή μονάδας εξαρτάται από τη γραμμή παραγγελίας.

Στον επίπεδο πίνακα όλα βρίσκονται μαζί, ενώ η μόνη εξάρτηση που θα έπρεπε να έχει μια γραμμή είναι από το κλειδί της. Η κανονικοποίηση χωρίζει τον πίνακα έτσι ώστε κάθε στήλη να εξαρτάται από το κλειδί του πίνακά της, από ολόκληρο το κλειδί και από τίποτα άλλο.



Σχήμα 18.1 · Ο επίπεδος πίνακας χωρίζεται σε πίνακες όπου κάθε στήλη εξαρτάται μόνο από το κλειδί του πίνακά της.

Οι κανονικές μορφές

Οι τρεις πρώτες κανονικές μορφές καλύπτουν σχεδόν όλα τα πρακτικά προβλήματα.

Η **πρώτη κανονική μορφή** ζητά ατομικές τιμές. Μια στήλη `phones` με τιμή `6944123456, 2105550101` παραβιάζει τον κανόνα, όπως και στήλες `phone1`, `phone2` και `phone3`. Δεν μπορείς να βρεις εύκολα έναν πελάτη από ένα τηλέφωνο, ούτε να βάλεις `UNIQUE` σε ένα τηλέφωνο. Η λύση είναι ένας πίνακας `customer_phones` με μία γραμμή ανά τηλέφωνο.

Η **δεύτερη κανονική μορφή** αφορά πίνακες με σύνθετο κλειδί. Καμία στήλη δεν πρέπει να εξαρτάται από μέρος μόνο του κλειδιού. Αν ο `order_items`, με κλειδί `(order_id, product_id)`, είχε και στήλη `product_name`, αυτή θα εξαρτιόταν μόνο από το `product_id`. Το όνομα του προϊόντος ανήκει στον `products`.

Η **τρίτη κανονική μορφή** απαγορεύει εξαρτήσεις μέσω άλλης στήλης. Αν ο `customers` είχε στήλες `city_name` και `region`, η περιφέρεια θα εξαρτιόταν από το κλειδί μόνο μέσω της πόλης. Η πόλη και η περιφέρεια ανήκουν στον `cities`.

Το σχήμα του βιβλίου είναι στην τρίτη κανονική μορφή, με δύο συνειδητές εξαιρέσεις που θα δούμε παρακάτω.

Σχέσεις ένα προς ένα, ένα προς πολλά και πολλά προς πολλά

Η σχέση **ένα προς πολλά** είναι η πιο συνηθισμένη. Ένας πελάτης, πολλές παραγγελίες. Το foreign key μπαίνει στην πλευρά του «πολλά», στον `orders`.

Η σχέση **πολλά προς πολλά** χρειάζεται τρίτο πίνακα, τον πίνακα σύνδεσης. Ο `product_suppliers` συνδέει προϊόντα και προμηθευτές και το κλειδί του είναι το ζεύγος. Ο πίνακας σύνδεσης μπορεί να έχει και δικές του στήλες, όπως η `supply_price`, που δεν ανήκει ούτε στο προϊόν ούτε στον προμηθευτή αλλά στη σχέση τους.

Η σχέση **ένα προς ένα** γράφεται με foreign key που είναι επίσης primary key ή unique. Είναι χρήσιμη όταν ένα κομμάτι των στοιχείων είναι προαιρετικό, μεγάλο ή με διαφορετικά δικαιώματα πρόσβασης.

SQL

```
CREATE TABLE customer_preferences (
  customer_id integer PRIMARY KEY REFERENCES customers (id),
  language    text NOT NULL DEFAULT 'el' CHECK (language IN ('el', 'en')),
  sms_opt_in  boolean NOT NULL DEFAULT false
);

INSERT INTO customer_preferences (customer_id, language) VALUES (21, 'en');
SELECT c.first_name, pref.language, pref.sms_opt_in
FROM customers AS c
JOIN customer_preferences AS pref ON pref.customer_id = c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 1

 first_name | language | sms_opt_in
------------+----------+------------
 Giorgos    | en       | f
(1 row)
```

Λίστες τιμών. CHECK, lookup table ή enum

Η στήλη `status` του `orders` δέχεται έξι τιμές. Η βάση το εγγυάται με `CHECK (status IN (...))`. Υπάρχουν τρεις τρόποι να περιορίσεις μια στήλη σε λίστα τιμών και ο καθένας ταιριάζει σε διαφορετική περίπτωση.

Το `CHECK` είναι το πιο απλό. Για να προσθέσεις τιμή αλλάζεις το constraint.

Ένα **lookup table** κρατά τις τιμές ως γραμμές, με foreign key από τη στήλη. Μπορεί να έχει και δικές του στήλες, όπως ελληνική ετικέτα ή σειρά ταξινόμησης. Νέα τιμή σημαίνει νέα γραμμή.

Ένας τύπος **enum** ορίζεται με `CREATE TYPE ... AS ENUM` και κρατά τις τιμές με συγκεκριμένη σειρά. Ταξινομείται κατά τη σειρά ορισμού και πιάνει λίγο χώρο. Προσθέτεις εύκολα τιμές, αλλά δεν αφαιρείς.

SQL

```
CREATE TYPE order_status AS ENUM
  ('pending', 'paid', 'shipped', 'delivered', 'cancelled', 'refunded');

SELECT 'paid'::order_status < 'shipped'::order_status AS earlier;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TYPE

 earlier
---------
 t
(1 row)
```

Για λίστες που αλλάζουν σπάνια και δεν έχουν δικά τους στοιχεία το `CHECK` αρκεί. Για λίστες που τις διαχειρίζονται χρήστες ή έχουν περιγραφές, το lookup table είναι το σωστό.

Πότε σπας τους κανόνες

Η κανονικοποίηση προστατεύει από ασυνέπειες. Υπάρχουν όμως δεδομένα που πρέπει να αντιγράφονται, γιατί η αντιγραφή είναι το ίδιο το γεγονός.

Η `unit_price` του `order_items` φαίνεται να αντιγράφει την `products.price`. Δεν την αντιγράφει. Καταγράφει την τιμή τη στιγμή της αγοράς. Αν τη διάβαζες από τον `products`, κάθε ανατίμηση θα άλλαζε τα τιμολόγια του παρελθόντος. Το ίδιο ισχύει για τη `shipping_city_id` του `orders`. Η διεύθυνση αποστολής μιας παραγγελίας δεν πρέπει να αλλάζει όταν ο πελάτης μετακομίσει.

Η δεύτερη περίπτωση είναι η απόδοση. Ένα σύνολο που υπολογίζεται από εκατομμύρια γραμμές σε κάθε σελίδα μπορεί να αποθηκευτεί. Αυτό είναι **denormalization** και έχει κόστος. Πρέπει να ξέρεις ποιος ενημερώνει το αντίγραφο και πότε. Το κεφάλαιο 20 δείχνει τα materialized views και το κεφάλαιο 39 τα triggers, δύο τρόπους να το κάνει η βάση για σένα.

Κλειδιά. Φυσικά ή τεχνητά

Ένα **φυσικό κλειδί** είναι στοιχείο του πραγματικού κόσμου, όπως το email ή το `sku`. Ένα **τεχνητό κλειδί** είναι ένας αριθμός χωρίς νόημα, όπως το `id`. Τα φυσικά κλειδιά αλλάζουν, γιατί οι άνθρωποι αλλάζουν email και οι κατάλογοι αλλάζουν κωδικούς. Αν ένα φυσικό κλειδί είναι primary key, κάθε αλλαγή του πρέπει να διαδοθεί σε όλα τα foreign keys.

Η συνηθισμένη πρακτική είναι τεχνητό primary key και `UNIQUE` constraint στο φυσικό κλειδί. Έτσι κρατάς και τη σταθερότητα του αριθμού και την εγγύηση μοναδικότητας του πραγματικού αναγνωριστικού. Ο `products` κάνει ακριβώς αυτό με το `id` και το `sku`.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Κάθε στήλη πρέπει να εξαρτάται από το κλειδί του πίνακά της, από ολόκληρο το κλειδί και από τίποτα άλλο. Οι σχέσεις πολλά προς πολλά χρειάζονται πίνακα σύνδεσης. Αντιγράφεις δεδομένα μόνο όταν η αντιγραφή καταγράφει ένα γεγονός ή όταν ξέρεις ποιος θα κρατά το αντίγραφο ενημερωμένο.

Ασκήσεις

ΑΣΚΗΣΗ 18.1 Εντοπισμός ασυνέπειας

Στον `flat_orders` της θεωρίας, βρες τις πόλεις που εμφανίζονται με περισσότερες από μία περιφέρειες. Εμφάνισε την πόλη και τις περιφέρειες σε μία στήλη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 city  |         regions
-------+-------------------------
 Χανιά | Κρήτη · Κρήτη και Νησιά
(1 row)
```

ΑΣΚΗΣΗ 18.2 Ανάκτηση του σχήματος

Από τον `flat_orders`, φτιάξε με `CREATE TABLE ... AS` έναν πίνακα `flat_cities` με μία γραμμή για κάθε ζεύγος πόλης και περιφέρειας. Εμφάνισέ τον ταξινομημένο κατά πόλη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 11

    city     |       region
-------------+--------------------
 Αθήνα       | Αττική
 Βόλος       | Θεσσαλία
 Ηράκλειο    | Κρήτη
 Θεσσαλονίκη | Κεντρική Μακεδονία
 Ιωάννινα    | Ήπειρος
 Καλαμάτα    | Πελοπόννησος
 Λάρισα      | Θεσσαλία
 Πάτρα       | Δυτική Ελλάδα
 Ρόδος       | Νότιο Αιγαίο
 Χανιά       | Κρήτη
 Χανιά       | Κρήτη και Νησιά
(11 rows)
```

ΑΣΚΗΣΗ 18.3 Lookup table για καταστάσεις

Φτιάξε πίνακα `order_statuses` με `code` primary key, ελληνική ετικέτα `label` και `sort_order` ακέραιο μοναδικό. Γέμισέ τον με τις έξι καταστάσεις και σύνδεσε τη στήλη `orders.status` με foreign key. Μετά εμφάνισε πλήθος παραγγελιών ανά κατάσταση με την ελληνική ετικέτα, κατά `sort_order`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 6

ALTER TABLE

    label    | orders
-------------+--------
 σε αναμονή  |      3
 πληρωμένη   |      5
 στον δρόμο  |     12
 παραδόθηκε  |    125
 ακυρώθηκε   |     14
 επιστράφηκε |      3
(6 rows)
```

ΑΣΚΗΣΗ 18.4 Πολλά τηλέφωνα

Φτιάξε πίνακα `customer_phones` με `customer_id`, `phone` και `kind` που δέχεται `κινητό`, `σταθερό` ή `εργασίας`. Κάθε τηλέφωνο να ανήκει σε ένα μόνο πελάτη. Μετέφερε τα υπάρχοντα τηλέφωνα των πελατών ως `κινητό` και εμφάνισε πόσα μεταφέρθηκαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 22

 phones
--------
     22
(1 row)
```

ΑΣΚΗΣΗ 18.5 Σύνοψη ως αντίγραφο

Φτιάξε με `CREATE TABLE ... AS` πίνακα `customer_stats` με `customer_id`, πλήθος παραγγελιών και ημερομηνία τελευταίας παραγγελίας. Εμφάνισε τη γραμμή του πελάτη 6. Μετά πρόσθεσε μια νέα παραγγελία στον πελάτη 6 και εμφάνισε ξανά τη γραμμή του.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 27

 customer_id | orders | last_order
-------------+--------+------------
           6 |     10 | 2026-06-13
(1 row)

INSERT 0 1

 customer_id | orders | last_order
-------------+--------+------------
           6 |     10 | 2026-06-13
(1 row)
```

ΚΕΦΑΛΑΙΟ 19

Transactions και savepoints

Ένα transaction ομαδοποιεί εντολές ώστε να γίνουν όλες ή καμία. Οι άλλες συνδέσεις δεν βλέπουν τις αλλαγές του μέχρι το `COMMIT`. Ένα σφάλμα μέσα στο transaction το ακυρώνει ολόκληρο, εκτός αν έχεις ορίσει savepoint.

Γιατί χρειάζονται

Μια νέα παραγγελία δεν είναι μία εντολή. Είναι μια γραμμή στον `orders`, γραμμές στον `order_items`, μείωση του αποθέματος και ίσως μια πληρωμή. Αν ο server κλείσει μετά το δεύτερο βήμα, η βάση θα έχει παραγγελία χωρίς πληρωμή και απόθεμα που δεν μειώθηκε. Χρειάζεσαι έναν τρόπο να πεις ότι αυτά τα βήματα είναι ένα.

BEGIN, COMMIT και ROLLBACK

Το `BEGIN` ξεκινά transaction. Το `COMMIT` το ολοκληρώνει και κάνει τις αλλαγές μόνιμες. Το `ROLLBACK` το ακυρώνει και η βάση επιστρέφει όπως ήταν πριν από το `BEGIN`.

```sql
BEGIN;
INSERT INTO orders (customer_id, status, created_at, shipping_city_id)
VALUES (30, 'paid', '2026-06-30 11:00', 1)
RETURNING id;
INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (163, 14, 1, 24.90), (163, 27, 1, 42.00);
UPDATE products SET stock = stock - 1 WHERE id IN (14, 27);
INSERT INTO payments (order_id, amount, method, paid_at)
VALUES (163, 66.90, 'card', '2026-06-30 11:01');
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

 id
-----
 163
(1 row)

INSERT 0 1

INSERT 0 2

UPDATE 2

INSERT 0 1

COMMIT
```

Οι τέσσερις εντολές έγιναν μαζί. Αν οποιαδήποτε αποτύγχανε, το `COMMIT` δεν θα έκανε μόνιμη καμία. Εδώ το `id` 163 γράφτηκε με το χέρι επειδή το είδαμε στο `RETURNING`. Σε εφαρμογή θα το έπαιρνες από το αποτέλεσμα της πρώτης εντολής. Στο κεφάλαιο 21 θα δεις πώς γίνονται όλα σε ένα query.

Χωρίς `BEGIN`, η PostgreSQL εκτελεί κάθε εντολή σε δικό της transaction. Αυτό λέγεται **autocommit**. Κάθε `INSERT` που έγραψες μέχρι τώρα ήταν ένα ολόκληρο transaction.

Το `ROLLBACK` είναι επίσης το καλύτερο δίχτυ ασφαλείας για επικίνδυνες αλλαγές. Γράφεις `BEGIN`, εκτελείς το `UPDATE`, ελέγχεις με `SELECT` και αποφασίζεις.

SQL

```sql
BEGIN;
UPDATE products SET price = price * 10;
SELECT name, price FROM products WHERE id = 1;
ROLLBACK;
SELECT name, price FROM products WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

UPDATE 30

                name                | price
------------------------------------+--------
 Βίος και πολιτεία του Αλέξη Ζορμπά | 169.00
(1 row)

ROLLBACK

                name                | price
------------------------------------+-------
 Βίος και πολιτεία του Αλέξη Ζορμπά | 16.90
(1 row)
```

Τι βλέπουν οι άλλοι

Όσο ένα transaction είναι ανοιχτό, οι αλλαγές του είναι ορατές μόνο στη δική του σύνδεση. Εδώ έχουμε δύο ταυτόχρονες συνδέσεις, Α και Β. Η Α προσθέτει μια πόλη χωρίς να κάνει `COMMIT`.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
INSERT INTO cities (id, name, region, population)
VALUES (12, 'Τρίπολη', 'Πελοπόννησος', 30912);
SELECT count(*) FROM cities;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

INSERT 0 1

 count
-------
    12
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
SELECT count(*) FROM cities;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
 count
-------
    11
(1 row)
```

Η Β βλέπει ακόμη 11 πόλεις. Η νέα γραμμή υπάρχει, αλλά δεν έχει γίνει commit. Μόλις η Α κάνει `COMMIT`, η Β τη βλέπει στο επόμενο query της.

SQL · ΣΥΝΕΔΡΙΑ Α

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
COMMIT
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
SELECT count(*) FROM cities;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
 count
-------
    12
(1 row)
```

Αυτή η απομόνωση είναι το **I** του ACID. Το κεφάλαιο 37 εξηγεί πώς την πετυχαίνει η PostgreSQL χωρίς να σταματά τους αναγνώστες και ποια επίπεδα απομόνωσης υπάρχουν.

Όταν μια εντολή αποτυγχάνει

Αν μια εντολή μέσα σε transaction αποτύχει, η PostgreSQL δεν ακυρώνει μόνο αυτή. Βάζει ολόκληρο το transaction σε κατάσταση σφάλματος και αρνείται κάθε επόμενη εντολή μέχρι το `ROLLBACK`.

SQL

```
BEGIN;
UPDATE products SET stock = stock + 5 WHERE id = 18;
INSERT INTO cities (id, name, region) VALUES (1, 'Αθήνα', 'Αττική');
SELECT stock FROM products WHERE id = 18;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

UPDATE 1

ERROR:  duplicate key value violates unique constraint "cities_pkey"
DETAIL:  Key (id)=(1) already exists.

ERROR:  current transaction is aborted, commands ignored until end of transaction block

ROLLBACK
```

Το `SELECT` απορρίφθηκε και το `COMMIT` έγινε στην πράξη `ROLLBACK`, όπως λέει η απάντηση. Η αύξηση του αποθέματος χάθηκε μαζί με την αποτυχημένη εισαγωγή. Είναι ακριβώς αυτό που θέλεις. Ένα transaction είναι όλα ή τίποτα.

> **ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL**
> Άλλες βάσεις ακυρώνουν μόνο την εντολή που απέτυχε και αφήνουν το transaction να συνεχίσει. Η PostgreSQL δεν το κάνει. Αν θέλεις αυτή τη συμπεριφορά, χρειάζεσαι savepoints. Το `psql` έχει τη ρύθμιση `\set ON_ERROR_ROLLBACK interactive`, που βάζει αυτόματα savepoint πριν από κάθε εντολή όταν πληκτρολογείς.

Savepoints

Ένα `SAVEPOINT όνομα` σημαδεύει ένα σημείο μέσα στο transaction. Το `ROLLBACK TO SAVEPOINT όνομα` ακυρώνει ό,τι έγινε μετά από αυτό και το transaction συνεχίζει κανονικά.

SQL

```
BEGIN;
UPDATE products SET stock = stock + 5 WHERE id = 18;
SAVEPOINT before_city;
INSERT INTO cities (id, name, region) VALUES (1, 'Αθήνα', 'Αττική');
ROLLBACK TO SAVEPOINT before_city;
SELECT stock FROM products WHERE id = 18;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

UPDATE 1

SAVEPOINT

ERROR:  duplicate key value violates unique constraint "cities_pkey"
DETAIL:  Key (id)=(1) already exists.

ROLLBACK

 stock
-------
     5
(1 row)

COMMIT
```

Η αποτυχημένη εισαγωγή ακυρώθηκε, αλλά η αύξηση του αποθέματος έμεινε και έγινε commit. Τα savepoints είναι χρήσιμα όταν ένα βήμα επιτρέπεται να αποτύχει, για παράδειγμα όταν φορτώνεις πολλές γραμμές και θέλεις να παραλείψεις όσες είναι λάθος.

Και το σχήμα είναι transactional

Στην PostgreSQL σχεδόν όλες οι εντολές σχήματος, όπως το `CREATE TABLE` και το `ALTER TABLE`, συμμετέχουν σε transactions. Μια αλλαγή σχήματος που αποτυγχάνει στη μέση δεν αφήνει τη βάση μισοαλλαγμένη.

SQL

```
BEGIN;
CREATE TABLE gift_messages (order_id integer PRIMARY KEY, message text);
ALTER TABLE orders ADD COLUMN gift boolean NOT NULL DEFAULT false;
ROLLBACK;
SELECT to_regclass('gift_messages') AS table_exists;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

CREATE TABLE

ALTER TABLE

ROLLBACK

 table_exists
--------------
 ø
(1 row)
```

Το `to_regclass` επιστρέφει `NULL` όταν ο πίνακας δεν υπάρχει. Ο πίνακας και η στήλη εξαφανίστηκαν με το `ROLLBACK`. Εξαίρεση είναι λίγες εντολές όπως το `CREATE INDEX CONCURRENTLY` και το `VACUUM`, που δεν τρέχουν μέσα σε transaction.

Deferred constraints

Ένα constraint ελέγχεται κανονικά μετά από κάθε εντολή. Μερικές φορές μια αλλαγή περνά από ενδιάμεση κατάσταση που παραβιάζει τον κανόνα, αλλά η τελική κατάσταση είναι σωστή. Η κλασική περίπτωση είναι η ανταλλαγή δύο μοναδικών τιμών.

SQL

```
CREATE TABLE shelf (
  product_id integer PRIMARY KEY REFERENCES products (id),
  position   integer NOT NULL,
  CONSTRAINT shelf_position_unique UNIQUE (position) DEFERRABLE INITIALLY IMMEDIATE
);
INSERT INTO shelf VALUES (1, 1), (2, 2);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 2
```

SQL

```
UPDATE shelf SET position = 2 WHERE product_id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  duplicate key value violates unique constraint "shelf_position_unique"
DETAIL:  Key ("position")=(2) already exists.
```

Το `DEFERRABLE` επιτρέπει στο transaction να αναβάλει τον έλεγχο μέχρι το `COMMIT` με την εντολή `SET CONSTRAINTS`.

SQL

```
BEGIN;
SET CONSTRAINTS shelf_position_unique DEFERRED;
UPDATE shelf SET position = 2 WHERE product_id = 1;
UPDATE shelf SET position = 1 WHERE product_id = 2;
COMMIT;
SELECT * FROM shelf ORDER BY position;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET CONSTRAINTS

UPDATE 1

UPDATE 1

COMMIT

 product_id | position
------------+----------
          2 |        1
          1 |        2
(2 rows)
```

Ανάμεσα στα δύο `UPDATE` δύο γραμμές είχαν θέση 2. Κανείς δεν το είδε, γιατί ο έλεγχος έγινε στο `COMMIT`, όταν η κατάσταση ήταν ξανά σωστή.

ACID

Οι τέσσερις εγγυήσεις ενός transaction έχουν συντομογραφία.

Γράμμα	Εγγύηση	Πώς την είδες
A · Atomicity	όλα ή τίποτα	`ROLLBACK` μετά από σφάλμα
C · Consistency	τα constraints ισχύουν μετά από κάθε commit	deferred constraints
I · Isolation	τα ταυτόχρονα transactions δεν βλέπουν μισές αλλαγές	οι δύο συνεδρίες
D · Durability	ό,τι έγινε commit επιζεί από crash	ο server γράφει στο WAL πριν απαντήσει

Το **WAL**, write ahead log, είναι ένα αρχείο στο οποίο η PostgreSQL γράφει κάθε αλλαγή πριν την εφαρμόσει στους πίνακες. Όταν το `COMMIT` επιστρέψει, η αλλαγή είναι ήδη στον δίσκο. Μετά από crash, η βάση ξαναπαίζει το WAL και φτάνει στην ίδια κατάσταση.

ΠΑΓΙΔΑ

Κράτα τα transactions σύντομα. Ένα transaction που μένει ανοιχτό κρατά locks σε γραμμές που άλλαξε και εμποδίζει τη βάση να καθαρίσει παλιές εκδόσεις γραμμών. Ποτέ μην περιμένεις απάντηση χρήστη ή κλήση σε εξωτερικό σύστημα μέσα σε ανοιχτό transaction. Τα κεφάλαια 29 και 38 δείχνουν τις συνέπειες.

ΚΡΑΤΑ ΑΥΤΟ

Το `BEGIN` ανοίγει transaction, το `COMMIT` κάνει τις αλλαγές μόνιμες και ορατές και το `ROLLBACK` τις ακυρώνει. Στην PostgreSQL ένα σφάλμα ακυρώνει ολόκληρο το transaction, εκτός αν επιστρέψεις σε savepoint. Και οι αλλαγές σχήματος είναι transactional.

Ασκήσεις

ΑΣΚΗΣΗ 19.1 Παραγγελία που δεν έγινε

Σε transaction, πρόσθεσε μια παραγγελία `pending` του πελάτη 26 στις `2026-06-30 12:00`. Βάλε μια γραμμή με το προϊόν 3 και ποσότητα 2 στην τιμή του προϊόντος, χρησιμοποιώντας το `currval` της ακολουθίας των παραγγελιών για το `order_id`. Εμφάνισε τη γραμμή, κάνε `ROLLBACK` και δείξε ότι ο πελάτης 26 δεν έχει παραγγελίες.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

INSERT 0 1

INSERT 0 1

 order_id | product_id | quantity | unit_price
----------+------------+----------+------------
      164 |          3 |        2 |       9.90
(1 row)

ROLLBACK

 orders
--------
      0
(1 row)
```

ΑΣΚΗΣΗ 19.2 Transaction σε σφάλμα

Ξεκίνα transaction, αύξησε κατά 10% την τιμή του προϊόντος 5 και μετά δοκίμασε να βάλεις προϊόν με `sku` `BK-PRG-001`, που υπάρχει ήδη. Δοκίμασε ένα `SELECT` της τιμής, κάνε `ROLLBACK` και εμφάνισε ξανά την τιμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

UPDATE 1

ERROR:  duplicate key value violates unique constraint "products_sku_key"
DETAIL:  Key (sku)=(BK-PRG-001) already exists.

ERROR:  current transaction is aborted, commands ignored until end of transaction block

ROLLBACK

 price
-------
 32.00
(1 row)
```

ΑΣΚΗΣΗ 19.3 Φόρτωση με savepoints

Σε transaction, πρόσθεσε τρεις κατηγορίες μία μία, με `id` 14, 13 και 15 και ονόματα `Παιχνίδια`, `Εργαλεία` και `Κήπος`, όλες με γονέα την 11. Βάλε savepoint πριν από κάθε εισαγωγή, ώστε η αποτυχία της δεύτερης, που έχει υπάρχον `id`, να μην ακυρώσει τις άλλες. Κάνε `COMMIT` και εμφάνισε τις κατηγορίες με `id` από 13 και πάνω.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SAVEPOINT

INSERT 0 1

SAVEPOINT

ERROR:  duplicate key value violates unique constraint "categories_pkey"
DETAIL:  Key (id)=(13) already exists.

ROLLBACK

SAVEPOINT

INSERT 0 1

COMMIT

 id |   name    | parent_id
----+-----------+-----------
 13 | Γραφείο   |        11
 14 | Παιχνίδια |        11
 15 | Κήπος     |        11
(3 rows)
```

ΑΣΚΗΣΗ 19.4 Αλλαγή σχήματος που αναιρείται

Σε transaction, πρόσθεσε στον `customers` στήλη `loyalty_points` ακέραιο με προεπιλογή 0 και δώσε 100 πόντους στον πελάτη 1. Εμφάνισε τους πόντους του, κάνε `ROLLBACK` και έλεγξε αν η στήλη υπάρχει ακόμη ρωτώντας τον κατάλογο `information_schema.columns`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

ALTER TABLE

UPDATE 1

 id | loyalty_points
----+----------------
  1 |            100
(1 row)

ROLLBACK

 columns_named_loyalty
-----------------------
                     0
(1 row)
```

ΑΣΚΗΣΗ 19.5 Ανταλλαγή θέσεων

Στον πίνακα `shelf` της θεωρίας, πρόσθεσε το προϊόν 3 στη θέση 3. Μετά, σε ένα transaction με deferred constraint, κάνε το προϊόν 3 πρώτο, το 1 δεύτερο και το 2 τρίτο. Εμφάνισε τον πίνακα κατά θέση.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1

BEGIN

SET CONSTRAINTS

UPDATE 1

UPDATE 1

UPDATE 1

COMMIT

 product_id | position
------------+----------
          3 |        1
          1 |        2
          2 |        3
(3 rows)
```

ΚΕΦΑΛΑΙΟ 20

Views και materialized views

Ένα view είναι ένα query με όνομα. Το χρησιμοποιείς σαν πίνακα, αλλά δεν κρατά δεδομένα και δείχνει πάντα την τρέχουσα κατάσταση. Ένα materialized view αποθηκεύει το αποτέλεσμα και είναι γρήγορο, με το κόστος ότι παλιώνει μέχρι να το ανανεώσεις.

CREATE VIEW

Η αξία μιας παραγγελίας χρειάστηκε σε πολλά κεφάλαια και κάθε φορά γράφαμε το ίδιο `sum(quantity * unit_price)` με `GROUP BY`. Ένα view δίνει όνομα σε αυτό το query.

SQL

```
CREATE VIEW order_totals AS
SELECT o.id AS order_id, o.customer_id, o.status, o.created_at,
       COALESCE(sum(oi.quantity * oi.unit_price), 0) AS total
FROM orders AS o
LEFT JOIN order_items AS oi ON oi.order_id = o.id
GROUP BY o.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE VIEW
```

Από εδώ και πέρα το `order_totals` χρησιμοποιείται σε κάθε query όπως ένας πίνακας.

SQL

```
SELECT c.first_name, c.last_name, sum(t.total) AS revenue
FROM order_totals AS t
JOIN customers AS c ON c.id = t.customer_id
WHERE t.status NOT IN ('cancelled', 'refunded')
GROUP BY c.id
ORDER BY revenue DESC
LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 first_name |  last_name   | revenue
------------+--------------+---------
 Σοφία      | Αλεξίου      | 7946.20
 Στέλιος    | Κουτσούκος   | 7291.60
 Γιώργος    | Παπαδόπουλος | 6630.00
(3 rows)
```

Το view δεν αποθηκεύει γραμμές. Κάθε φορά που το χρησιμοποιείς, η PostgreSQL αντικαθιστά το όνομά του με το query του και σχεδιάζει το συνολικό query από την αρχή. Γι’ αυτό ένα view δεν είναι

ούτε πιο αργό ούτε πιο γρήγορο από το query που περιέχει. Και γι' αυτό δείχνει πάντα τα τρέχοντα δεδομένα.

```sql
INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (162, 22, 1, 249.00);

SELECT order_id, status, total FROM order_totals WHERE order_id = 162;
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1

 order_id | status  | total
----------+---------+--------
      162 | pending | 249.00
(1 row)
```

Η παραγγελία 162 είχε αξία 0 και τώρα έχει 249. Δεν χρειάστηκε καμία ενημέρωση του view.

Τι κερδίζεις

Ένα view δίνει τρία πράγματα. Πρώτα, επαναχρησιμοποίηση. Ο ορισμός της αξίας παραγγελίας γράφεται σε ένα σημείο και αν αλλάξει, για παράδειγμα για να αφαιρεί εκπτώσεις κουπονιών, αλλάζει παντού μαζί.

Δεύτερο, απλούστερη διεπαφή. Ένα query αναφοράς διαβάζει `order_totals` αντί για joins και `GROUP BY`.

Τρίτο, έλεγχο πρόσβασης. Ένας χρήστης μπορεί να έχει δικαίωμα να διαβάζει ένα view που δείχνει μόνο όνομα και πόλη πελατών, χωρίς δικαίωμα στον πίνακα `customers` με τα email και τα τηλέφωνα. Το κεφάλαιο 41 επιστρέφει σε αυτό.

Αλλαγή και διαγραφή

Το `CREATE OR REPLACE VIEW` αλλάζει τον ορισμό ενός view. Μπορεί να προσθέσει στήλες στο τέλος, αλλά δεν μπορεί να αφαιρέσει ή να μετονομάσει υπάρχουσες. Για τέτοιες αλλαγές κάνεις `DROP VIEW` και `CREATE VIEW`.

Η βάση ξέρει ποια views εξαρτώνται από ποιους πίνακες και στήλες. Δεν σε αφήνει να σβήσεις κάτι που χρειάζεται ένα view.

```sql
ALTER TABLE order_items DROP COLUMN unit_price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  cannot drop column unit_price of table order_items because other objects depend on it
DETAIL:  view order_totals depends on column unit_price of table order_items
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

Το `DROP ... CASCADE` θα έσβηνε και το view. Χρησιμοποίησέ το μόνο όταν ξέρεις ποια αντικείμενα θα σβηστούν μαζί. Το μήνυμα σφάλματος τα αναφέρει.

Views που δέχονται αλλαγές

Ένα απλό view, με έναν πίνακα στο `FROM` και χωρίς `GROUP BY`, `DISTINCT` ή aggregates, δέχεται `INSERT`, `UPDATE` και `DELETE`. Η PostgreSQL μεταφράζει την αλλαγή σε αλλαγή του πίνακα.

SQL

```
CREATE VIEW active_products AS
SELECT id, sku, name, category_id, price, stock, active
FROM products
WHERE active
WITH CHECK OPTION;

UPDATE active_products SET stock = stock + 1 WHERE sku = 'PR-005'
RETURNING sku, stock;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE VIEW

  sku   | stock
--------+-------
 PR-005 |     1
(1 row)

UPDATE 1
```

Το `WITH CHECK OPTION` απαγορεύει αλλαγές που θα έβγαζαν τη γραμμή από το view. Χωρίς αυτό, ένα `INSERT` μέσω του view θα μπορούσε να βάλει ανενεργό προϊόν, που μετά δεν θα εμφανιζόταν στο ίδιο view.

SQL

```
UPDATE active_products SET active = false WHERE sku = 'PR-005';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  new row violates check option for view "active_products"
DETAIL:  Failing row contains (18, PR-005, Webcam HD, 9, 59.00, 30.00, 1, f,
{περιφερειακά,video}, {"microphone": true, "resolution": "1920x1080"}, Κάμερα Full HD
για τηλεδιασκέψεις.).
```

Materialized views

Ένα view υπολογίζει το query του κάθε φορά. Για ένα βαρύ query σε μεγάλους πίνακες, που εμφανίζεται σε κάθε φόρτωση μιας σελίδας, αυτό μπορεί να κοστίζει. Ένα **materialized view** εκτελεί το query μία φορά και αποθηκεύει το αποτέλεσμα σαν πίνακα.

```sql
CREATE MATERIALIZED VIEW monthly_revenue AS
SELECT date_trunc('month', created_at)::date AS month,
       count(*) AS orders,
       sum(total) AS revenue
FROM order_totals
WHERE status NOT IN ('cancelled', 'refunded')
GROUP BY 1;

SELECT * FROM monthly_revenue WHERE month >= '2026-04-01' ORDER BY month;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 18

   month    | orders | revenue
------------+--------+----------
 2026-04-01 |      8 |  4026.50
 2026-05-01 |     18 | 10338.80
 2026-06-01 |     29 |  9323.90
(3 rows)
```

Το `GROUP BY 1` ομαδοποιεί με την πρώτη στήλη της λίστας. Το κεφάλαιο 4 το απέφυγε για το `ORDER BY`, αλλά σε ένα σύντομο ορισμό όπως αυτός διαβάζεται εύκολα. Ένα materialized view μπορεί να βασίζεται σε κανονικό view, όπως εδώ στο `order_totals`.

Το κόστος είναι ότι τα δεδομένα του παλιώνουν. Μια νέα πληρωμένη παραγγελία δεν εμφανίζεται μέχρι να ζητήσεις ανανέωση με `REFRESH MATERIALIZED VIEW`.

```sql
INSERT INTO orders (customer_id, status, created_at)
VALUES (30, 'paid', '2026-06-30 10:00');
INSERT INTO order_items (order_id, product_id, quantity, unit_price)
VALUES (currval(pg_get_serial_sequence('orders', 'id')), 12, 1, 2199.00);

SELECT month, orders, revenue FROM monthly_revenue WHERE month = '2026-06-01';
REFRESH MATERIALIZED VIEW monthly_revenue;
SELECT month, orders, revenue FROM monthly_revenue WHERE month = '2026-06-01';
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1

INSERT 0 1

   month    | orders | revenue
------------+--------+----------
 2026-06-01 |     29 | 9323.90
(1 row)

REFRESH MATERIALIZED VIEW

   month    | orders | revenue
------------+--------+----------
 2026-06-01 |     30 | 11522.90
(1 row)
```

Το `REFRESH` ξαναεκτελεί το query και αντικαθιστά τα δεδομένα. Όσο διαρκεί, κλειδώνει το materialized view και κανείς δεν μπορεί να το διαβάσει. Το `REFRESH MATERIALIZED VIEW CONCURRENTLY` επιτρέπει αναγνώσεις κατά την ανανέωση, αλλά χρειάζεται unique index στο materialized view, ώστε να μπορεί να αντιστοιχίσει παλιές και νέες γραμμές.

SQL

```
CREATE UNIQUE INDEX monthly_revenue_month ON monthly_revenue (month);
REFRESH MATERIALIZED VIEW CONCURRENTLY monthly_revenue;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

REFRESH MATERIALIZED VIEW
```

ΠΑΓΙΔΑ

Η PostgreSQL δεν ανανεώνει ποτέ μόνη της ένα materialized view. Κάποιος πρέπει να τρέχει το `REFRESH`, από ένα cron job, από την εφαρμογή ή από trigger. Πριν φτιάξεις ένα, αποφάσισε πόσο παλιά δεδομένα αντέχει όποιος το διαβάζει και ποιος θα το ανανεώνει.

ΚΡΑΤΑ ΑΥΤΟ

Ένα view είναι ένα αποθηκευμένο query. Δεν κρατά δεδομένα, δείχνει πάντα την τρέχουσα κατάσταση και έχει την απόδοση του query του. Ένα materialized view αποθηκεύει το αποτέλεσμα, διαβάζεται γρήγορα και ανανεώνεται μόνο με `REFRESH`.

Ασκήσεις

ΑΣΚΗΣΗ 20.1 Επαφές πελατών

Φτιάξε view `customer_contacts` με `id`, ονοματεπώνυμο ως `full_name`, όνομα πόλης ή `άγνωστη` ως `city` και το πρώτο διαθέσιμο από email και τηλέφωνο ως `contact`. Εμφάνισε τους πελάτες 8 έως 12 από το view.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE VIEW

 id |      full_name       |   city   |          contact
----+----------------------+----------+---------------------------
  8 | Παναγιώτης Βασιλείου | Αθήνα    | p.vasileiou@example.gr
  9 | Σοφία Αλεξίου        | Βόλος    | sofia.alexiou@example.gr
 10 | Χρήστος Μιχαηλίδης   | άγνωστη  | christos.m@example.gr
 11 | Ιωάννα Παππά         | Ιωάννινα | ioanna.pappa@example.gr
 12 | Αλέξανδρος Οικονόμου | Αθήνα    | alex.oikonomou@example.gr
(5 rows)
```

ΑΣΚΗΣΗ 20.2 Μέση αξία ανά κατάσταση

Χρησιμοποιώντας το view `order_totals` της θεωρίας, εμφάνισε για κάθε κατάσταση το πλήθος των παραγγελιών και τη μέση αξία τους με δύο δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  status   | orders | avg_total
-----------+--------+-----------
 paid      |      6 |    600.72
 pending   |      3 |    548.07
 refunded  |      3 |    539.67
 delivered |    125 |    478.62
 cancelled |     14 |    252.00
 shipped   |     12 |    167.06
(6 rows)
```

ΑΣΚΗΣΗ 20.3 Εισαγωγή μέσα από view

Μέσα από το `active_products`, πρόσθεσε το προϊόν `PR-007` με όνομα `Hub USB`, κατηγορία 9, τιμή 29, απόθεμα 40 και `active` αληθές. Μετά δοκίμασε να προσθέσεις το `PR-008` με `active` ψευδές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |  sku
----+--------
 31 | PR-007
(1 row)

INSERT 0 1

ERROR:  new row violates check option for view "active_products"
DETAIL:  Failing row contains (32, PR-008, Καλώδιο HDMI, 9, 9.00, null, 100, f, {}, {}, null).
```

ΑΣΚΗΣΗ 20.4 Κορυφαία προϊόντα ως materialized view

Φτιάξε materialized view `product_sales` με `product_id`, όνομα, συνολικά τεμάχια και έσοδα από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν. Εμφάνισε τα τρία προϊόντα με τα μεγαλύτερα έσοδα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 29

       name        | units | revenue
-------------------+-------+----------
 Laptop Pro 16     |    11 | 23989.00
 Laptop Student 15 |    11 |  7139.00
 Laptop Aero 14    |     6 |  6844.00
(3 rows)
```

ΑΣΚΗΣΗ 20.5 Παλιά δεδομένα και ανανέωση

Στο `product_sales` της προηγούμενης άσκησης, πρόσθεσε μια πληρωμένη παραγγελία του πελάτη 26 με τρία `Ασύρματο ποντίκι`. Εμφάνισε τη γραμμή του προϊόντος από το materialized view πριν και μετά από `REFRESH`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 29

INSERT 0 1

INSERT 0 1

 units | revenue
-------+---------
     9 |  224.10
(1 row)

REFRESH MATERIALIZED VIEW

 units | revenue
-------+---------
    12 |  298.80
(1 row)
```

ΜΕΡΟΣ IV

Advanced SQL

Εργαλεία που μετατρέπουν ένα query από μια λίστα φίλτρων σε πρόγραμμα. CTE, αναδρομή, window functions, πολυδιάστατες συνόψεις, JSONB και αναζήτηση κειμένου με trigrams.

ΚΕΦΑΛΑΙΟ 21

CTE με WITH

Ένα CTE δίνει όνομα σε ένα ενδιάμεσο αποτέλεσμα και το τοποθετεί πριν από το κύριο query. Ένα μεγάλο query διαβάζεται τότε από πάνω προς τα κάτω, βήμα βήμα. Στην PostgreSQL ένα CTE μπορεί επίσης να αλλάζει δεδομένα.

WITH

Στο κεφάλαιο 11 διορθώσαμε το fan out με δύο derived tables μέσα στο `FROM`. Το query δούλευε, αλλά για να το καταλάβεις έπρεπε να το διαβάσεις από μέσα προς τα έξω. Ένα **CTE**, common table expression, γράφει τα ίδια βήματα στην αρχή, με όνομα.

SQL

```
WITH item_totals AS (
  SELECT order_id, sum(quantity * unit_price) AS items_total
  FROM order_items
  GROUP BY order_id
),
paid_totals AS (
  SELECT order_id, sum(amount) AS paid_total
  FROM payments
  GROUP BY order_id
)
SELECT o.id, it.items_total, pt.paid_total
FROM orders AS o
JOIN item_totals AS it ON it.order_id = o.id
JOIN paid_totals AS pt ON pt.order_id = o.id
WHERE o.customer_id = 13
ORDER BY o.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id | items_total | paid_total
----+-------------+------------
 20 |      114.90 |     114.90
 35 |      690.80 |     690.80
 40 |     2527.50 |    2527.50
 41 |      289.00 |     289.00
 82 |      164.00 |     164.00
(5 rows)
```

Το `WITH` ακολουθείται από ένα ή περισσότερα ονόματα, το καθένα με ένα query σε παρενθέσεις, χωρισμένα με κόμμα. Μετά έρχεται το κύριο query, που χρησιμοποιεί τα ονόματα σαν πίνακες. Τα CTE υπάρχουν μόνο για αυτό το query.

Βήματα που χτίζουν το ένα πάνω στο άλλο

Ένα CTE μπορεί να χρησιμοποιεί τα προηγούμενα. Έτσι ένα πρόβλημα με πολλά βήματα γράφεται με τη σειρά που το σκέφτεσαι. Οι πελάτες που ξόδεψαν περισσότερα από τον μέσο πελάτη βρίσκονται σε τρία βήματα.

SQL

```
WITH order_value AS (
  SELECT o.id, o.customer_id, sum(oi.quantity * oi.unit_price) AS total
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  WHERE o.status NOT IN ('cancelled', 'refunded')
  GROUP BY o.id
),
customer_value AS (
  SELECT customer_id, sum(total) AS spent, count(*) AS orders
  FROM order_value
  GROUP BY customer_id
),
average AS (
  SELECT avg(spent) AS avg_spent FROM customer_value
)
SELECT cv.customer_id, cv.orders, cv.spent, round(a.avg_spent, 2) AS avg_spent
FROM customer_value AS cv
CROSS JOIN average AS a
WHERE cv.spent > 1.5 * a.avg_spent
ORDER BY cv.spent DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | orders |  spent  | avg_spent
-------------+--------+---------+-----------
           9 |      7 | 7946.20 |   2393.82
          23 |      7 | 7291.60 |   2393.82
           2 |      8 | 6630.00 |   2393.82
          14 |      6 | 5177.50 |   2393.82
           8 |      7 | 4401.80 |   2393.82
          13 |      5 | 3786.20 |   2393.82
(6 rows)
```

Κάθε βήμα έχει όνομα που λέει τι περιέχει. Αν ένα αποτέλεσμα φαίνεται λάθος, αντικαθιστάς το κύριο query με `SELECT * FROM customer_value` και ελέγχεις το ενδιάμεσο βήμα. Αυτή η δυνατότητα αποσφαλμάτωσης είναι από μόνη της λόγος να γράφεις μεγάλα queries με CTE.

Ένα CTE μπορεί να χρησιμοποιηθεί και πολλές φορές στο ίδιο query. Εδώ κάθε μήνας συγκρίνεται με τον καλύτερο μήνα, που υπολογίζεται από το ίδιο CTE.

SQL

```
WITH monthly AS (
  SELECT date_trunc('month', o.created_at)::date AS month,
         sum(oi.quantity * oi.unit_price) AS revenue
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  WHERE o.status NOT IN ('cancelled', 'refunded')
    AND o.created_at >= '2026-01-01'
  GROUP BY 1
)
SELECT m.month, m.revenue,
       round(100 * m.revenue / (SELECT max(revenue) FROM monthly), 1) AS pct_of_best
FROM monthly AS m
ORDER BY m.month;
```

ΑΠΟΤΕΛΕΣΜΑ

```
   month    | revenue  | pct_of_best
------------+----------+-------------
 2026-01-01 |  1860.80 |        18.0
 2026-02-01 |  3747.20 |        36.2
 2026-03-01 |  1599.20 |        15.5
 2026-04-01 |  4026.50 |        38.9
 2026-05-01 | 10338.80 |       100.0
 2026-06-01 |  9074.90 |        87.8
(6 rows)
```

Πώς εκτελείται ένα CTE

Από την PostgreSQL 12, ένα CTE που χρησιμοποιείται μία φορά και δεν αλλάζει δεδομένα ενσωματώνεται στο κύριο query, όπως ένα derived table. Ο planner μπορεί να σπρώξει φίλτρα του κύριου query μέσα του και να χρησιμοποιήσει indexes. Ένα CTE που χρησιμοποιείται πολλές φορές υπολογίζεται μία φορά και το αποτέλεσμά του κρατιέται σε μνήμη.

Μπορείς να ορίσεις ρητά τη συμπεριφορά με `AS MATERIALIZED` ή `AS NOT MATERIALIZED`. Το πρώτο αναγκάζει την PostgreSQL να υπολογίσει το CTE ολόκληρο πριν από το κύριο query. Είναι χρήσιμο όταν ένα ακριβό CTE χρησιμοποιείται πολλές φορές ή όταν θέλεις να εμποδίσεις τον planner να ενσωματώσει ένα βήμα. Θα δεις τη διαφορά στα plans του κεφαλαίου 31.

> **ΠΑΓΙΔΑ**
> Σε παλαιότερες εκδόσεις, πριν από την 12, κάθε CTE υπολογιζόταν ολόκληρο πριν από το κύριο query. Θα διαβάσεις ακόμη συμβουλές ότι τα CTE είναι αργά επειδή εμποδίζουν τα φίλτρα να περάσουν μέσα τους. Για σύγχρονες εκδόσεις δεν ισχύει, εκτός αν γράψεις `MATERIALIZED`.

CTE που αλλάζουν δεδομένα

Στην PostgreSQL ένα CTE μπορεί να είναι `INSERT`, `UPDATE` ή `DELETE` με `RETURNING`. Οι γραμμές του `RETURNING` γίνονται το αποτέλεσμα του CTE και μπορούν να τροφοδοτήσουν την επόμενη εντολή. Έτσι λύνεται το πρόβλημα του κεφαλαίου 19, όπου το `id` της νέας παραγγελίας γράφτηκε με το χέρι.

SQL

```
WITH new_order AS (
  INSERT INTO orders (customer_id, status, created_at, shipping_city_id)
  VALUES (30, 'paid', '2026-06-30 11:00', 1)
  RETURNING id
),
lines AS (
  INSERT INTO order_items (order_id, product_id, quantity, unit_price)
  SELECT new_order.id, p.id, 1, p.price
  FROM new_order
  CROSS JOIN products AS p
  WHERE p.id IN (14, 27)
  RETURNING order_id, unit_price
)
INSERT INTO payments (order_id, amount, method, paid_at)
SELECT order_id, sum(unit_price), 'card', '2026-06-30 11:01'
FROM lines
GROUP BY order_id
RETURNING order_id, amount;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | amount
----------+--------
      163 |  66.90
(1 row)

INSERT 0 1
```

Τρεις εντολές έγιναν σε ένα query. Κάθε query είναι ένα transaction, οπότε είτε μπαίνουν και τα τρία είτε κανένα. Το ίδιο μοτίβο μετακινεί γραμμές από έναν πίνακα σε άλλον, με `DELETE ... RETURNING` σε CTE και `INSERT ... SELECT` στο κύριο query.

Όλα τα κομμάτια ενός `WITH` βλέπουν την ίδια εικόνα της βάσης, αυτή που υπήρχε πριν αρχίσει το query. Ένα `SELECT` στο κύριο query δεν βλέπει τις αλλαγές που έκανε ένα CTE στον ίδιο πίνακα. Βλέπει μόνο όσα επιστρέφει το `RETURNING`.

SQL

```
WITH sold AS (
  UPDATE products SET stock = stock - 1 WHERE id = 1
  RETURNING stock
)
SELECT (SELECT stock FROM sold) AS from_returning,
       (SELECT stock FROM products WHERE id = 1) AS from_table;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 from_returning | from_table
----------------+------------
             41 |         42
(1 row)
```

Το `from_table` δείχνει το απόθεμα πριν από την αλλαγή. Αν θέλεις τη νέα τιμή, τη διαβάζεις από το `RETURNING`.

ΚΡΑΤΑ ΑΥΤΟ

Το `WITH` δίνει όνομα σε ενδιάμεσα βήματα και κάνει ένα μεγάλο query ευανάγνωστο από πάνω προς τα κάτω. Από την PostgreSQL 12 τα CTE ενσωματώνονται στο κύριο query όταν χρησιμοποιούνται μία φορά. Ένα CTE με `INSERT`, `UPDATE` ή `DELETE` και `RETURNING` επιτρέπει πολλές αλλαγές σε ένα ατομικό query.

Ασκήσεις

ΑΣΚΗΣΗ 21.1 Πληρωμές ανά περιφέρεια

Με δύο CTE, υπολόγισε πρώτα το καθαρό ποσό που πλήρωσε κάθε πελάτης και μετά άθροισέ το ανά περιφέρεια κατοικίας. Εμφάνισε περιφέρεια, πλήθος πελατών με πληρωμές και σύνολο, από το μεγαλύτερο σύνολο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       region        | customers |   paid
---------------------+-----------+----------
 Αττική              |         7 | 18233.60
 Κεντρική Μακεδονία  |         6 | 15901.30
 Θεσσαλία            |         4 | 10167.30
 Κρήτη               |         5 |  9150.40
 Δυτική Ελλάδα       |         2 |  5092.60
 Ήπειρος             |         1 |  2101.80
 Πελοπόννησος        |         1 |  1158.60
 Νότιο Αιγαίο        |         1 |   662.50
(8 rows)
```

ΑΣΚΗΣΗ 21.2 Πάνω από τον μέσο όρο της κατηγορίας τους

Με CTE, υπολόγισε τα έσοδα κάθε προϊόντος από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν και τον μέσο όρο των εσόδων ανά κατηγορία. Εμφάνισε τα προϊόντα με έσοδα πάνω από τον μέσο όρο της κατηγορίας τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                name                 | revenue  | category_avg
-------------------------------------+----------+--------------
 Βίος και πολιτεία του Αλέξη Ζορμπά |   253.50 |       192.78
 Ματωμένα χώματα                     |   228.00 |       192.78
 Μαθαίνω Python                      |   541.00 |       430.67
 Καθαρός κώδικας στην πράξη          |   456.00 |       430.67
 PostgreSQL σε βάθος                 |   585.00 |       562.50
 Laptop Pro 16                       | 21790.00 |      9143.00
 Οθόνη 27" 4K                        |  3948.00 |      1340.00
 Ακουστικά ANC                       |  2647.00 |      1661.67
 Μικρόφωνο USB                       |  1785.00 |      1661.67
 Καφετιέρα espresso                  |  2241.00 |      1173.63
 Εργονομική καρέκλα                  |  4624.00 |      2947.33
 Γραφείο ρυθμιζόμενου ύψους          |  3672.00 |      2947.33
(12 rows)
```

ΑΣΚΗΣΗ 21.3 Παραγγελία σε ένα query

Με data modifying CTE, πρόσθεσε σε ένα query μια παραγγελία `pending` του πελάτη 26 στις `2026-06-30 14:00` με δύο γραμμές, τα προϊόντα 5 και 6 σε ποσότητα 1 και στην τρέχουσα τιμή τους. Επέστρεψε το `order_id`, το `product_id` και την τιμή των γραμμών.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | product_id | unit_price
----------+------------+------------
      164 |          5 |      32.00
      164 |          6 |      29.50
(2 rows)

INSERT 0 2
```

ΑΣΚΗΣΗ 21.4 Αρχειοθέτηση κριτικών

Φτιάξε πίνακα `reviews_archive` με την ίδια δομή με τον `reviews`, με `CREATE TABLE reviews_archive (LIKE reviews)`. Μετακίνησε σε αυτόν με ένα query τις κριτικές που γράφτηκαν πριν από τον Ιούλιο του 2025 και επέστρεψε το πλήθος τους. Μετά εμφάνισε πόσες κριτικές έμειναν στον `reviews`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 23

 archived | remaining
----------+-----------
       23 |        16
(1 row)
```

ΑΣΚΗΣΗ 21.5 Τι βλέπει το κύριο query

Σε ένα query με CTE, αύξησε κατά 5 το απόθεμα των προϊόντων της κατηγορίας 13 επιστρέφοντας `id` και νέο απόθεμα. Στο κύριο query εμφάνισε για κάθε προϊόν το όνομα από τον `products`, το απόθεμα όπως το βλέπει ο πίνακας και το απόθεμα από το `RETURNING`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             | table_stock | returned_stock
-----------------------------+-------------+----------------
 Εργονομική καρέκλα          |           6 |             11
 Γραφείο ρυθμιζόμενου ύψους  |           3 |              8
 Φωτιστικό γραφείου LED      |          27 |             32
(3 rows)
```

ΚΕΦΑΛΑΙΟ 22

Recursive CTE για δέντρα και γράφους

Ένα recursive CTE χρησιμοποιεί το δικό του αποτέλεσμα για να παράγει νέες γραμμές, μέχρι να μην παράγει άλλες. Είναι ο τρόπος να διασχίσεις μια ιεραρχία άγνωστου βάθους ή να βρεις διαδρομές σε έναν γράφο με ένα query.

Η δομή

Στο κεφάλαιο 9 κάθε επίπεδο της ιεραρχίας των υπαλλήλων χρειαζόταν ένα ακόμη self join. Για άγνωστο βάθος χρειάζεσαι επανάληψη. Ένα recursive CTE έχει δύο κομμάτια ενωμένα με `UNION ALL`.

SQL

```
WITH RECURSIVE numbers AS (
  SELECT 1 AS n
  UNION ALL
  SELECT n + 1 FROM numbers WHERE n < 5
)
SELECT n FROM numbers;
```

ΑΠΟΤΕΛΕΣΜΑ

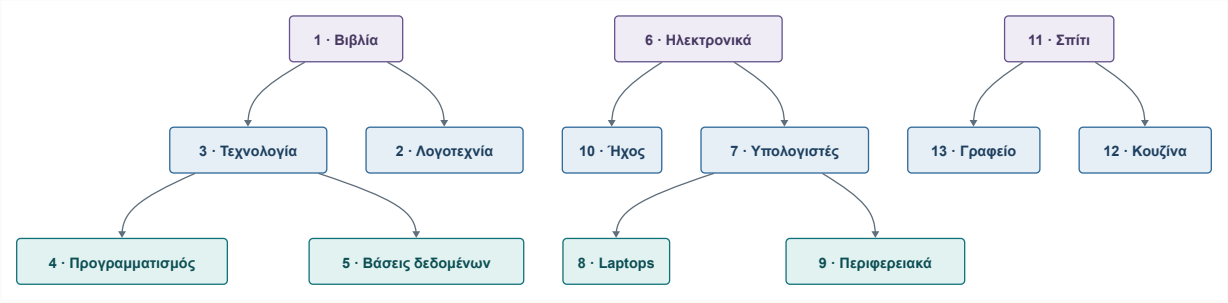
```
 n
---
 1
 2
 3
 4
 5
(5 rows)
```

Το πρώτο κομμάτι είναι το **anchor**. Εκτελείται μία φορά και δίνει τις αρχικές γραμμές. Το δεύτερο είναι ο **αναδρομικός όρος**. Αναφέρεται στο ίδιο το CTE, αλλά σε κάθε βήμα βλέπει μόνο τις γραμμές που παρήγαγε το προηγούμενο βήμα. Η PostgreSQL επαναλαμβάνει τον αναδρομικό όρο μέχρι να επιστρέψει μηδέν γραμμές. Το αποτέλεσμα είναι η ένωση όλων των βημάτων.

Στο παράδειγμα, το anchor δίνει το 1. Το πρώτο βήμα βλέπει το 1 και δίνει το 2. Το τέταρτο βλέπει το 4 και δίνει το 5. Το πέμπτο βλέπει το 5, το `WHERE n < 5` το απορρίπτει και η αναδρομή σταματά. Για αριθμούς το `generate_series` είναι απλούστερο. Η αναδρομή αξίζει όταν κάθε βήμα χρειάζεται δεδομένα από πίνακα.

Διάσχιση δέντρου προς τα κάτω

Οι κατηγορίες είναι δέντρο. Κάθε κατηγορία έχει `parent_id` που δείχνει στη γονική της.



Σχήμα 22.1 · Το δέντρο των κατηγοριών. Τρεις ρίζες χωρίς γονέα και έως δύο επίπεδα από κάτω.

Όλες οι υποκατηγορίες της κατηγορίας Βιβλία, σε όσο βάθος κι αν βρίσκονται, είναι οι γραμμές που φτάνεις ξεκινώντας από την 1 και ακολουθώντας συνεχώς το `parent_id` ανάποδα.

SQL

```
WITH RECURSIVE tree AS (
  SELECT id, name, parent_id, 0 AS depth
  FROM categories
  WHERE id = 1
  UNION ALL
  SELECT c.id, c.name, c.parent_id, t.depth + 1
  FROM categories AS c
  JOIN tree AS t ON c.parent_id = t.id
)
SELECT id, repeat('  ', depth) || name AS category, depth
FROM tree;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |       category        | depth
----+-----------------------+-------
  1 | Βιβλία                |     0
  2 |   Λογοτεχνία          |     1
  3 |   Τεχνολογία          |     1
  4 |     Προγραμματισμός   |     2
  5 |     Βάσεις δεδομένων  |     2
(5 rows)
```

Σε κάθε βήμα το join βρίσκει τα παιδιά των γραμμών του προηγούμενου βήματος. Το `depth` αυξάνεται κατά ένα σε κάθε επίπεδο. Όταν ένα βήμα δεν βρίσκει παιδιά, η αναδρομή τελειώνει.

Μονοπάτια και ταξινόμηση

Η σειρά του αποτελέσματος ακολουθεί τα βήματα. Πρώτα όλο το επίπεδο 1, μετά όλο το επίπεδο 2. Για να εμφανίζεται κάθε κατηγορία κάτω από τη γονική της χρειάζεσαι μια στήλη ταξινόμησης. Η πιο απλή είναι το μονοπάτι από τη ρίζα.

SQL

```
WITH RECURSIVE tree AS (
  SELECT id, name, name AS path, 0 AS depth
  FROM categories
  WHERE parent_id IS NULL
  UNION ALL
  SELECT c.id, c.name, t.path || ' / ' || c.name, t.depth + 1
  FROM categories AS c
  JOIN tree AS t ON c.parent_id = t.id
)
SELECT repeat('  ', depth) || name AS category, path
FROM tree
ORDER BY path;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       category       |                    path
----------------------+-----------------------------------------
 Βιβλία                | Βιβλία
   Λογοτεχνία          | Βιβλία / Λογοτεχνία
   Τεχνολογία          | Βιβλία / Τεχνολογία
     Βάσεις δεδομένων  | Βιβλία / Τεχνολογία / Βάσεις δεδομένων
     Προγραμματισμός   | Βιβλία / Τεχνολογία / Προγραμματισμός
 Ηλεκτρονικά           | Ηλεκτρονικά
   Ήχος                | Ηλεκτρονικά / Ήχος
   Υπολογιστές         | Ηλεκτρονικά / Υπολογιστές
     Περιφερειακά      | Ηλεκτρονικά / Υπολογιστές / Περιφερειακά
     Laptops           | Ηλεκτρονικά / Υπολογιστές / Laptops
 Σπίτι                 | Σπίτι
   Γραφείο             | Σπίτι / Γραφείο
   Κουζίνα             | Σπίτι / Κουζίνα
(13 rows)
```

Το anchor ξεκινά τώρα από όλες τις ρίζες, `parent_id IS NULL`, οπότε το αποτέλεσμα είναι ολόκληρο το δέντρο. Η PostgreSQL έχει και μια ειδική σύνταξη για αυτό, το `SEARCH DEPTH FIRST BY`, που φτιάχνει τη στήλη ταξινόμησης αυτόματα.

SQL

```
WITH RECURSIVE tree AS (
  SELECT id, name, 0 AS depth FROM categories WHERE parent_id IS NULL
  UNION ALL
  SELECT c.id, c.name, t.depth + 1
  FROM categories AS c
  JOIN tree AS t ON c.parent_id = t.id
) SEARCH DEPTH FIRST BY name SET ordering
SELECT repeat('  ', depth) || name AS category
FROM tree
ORDER BY ordering;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       category
----------------------
 Βιβλία
   Λογοτεχνία
   Τεχνολογία
     Βάσεις δεδομένων
     Προγραμματισμός
 Ηλεκτρονικά
   Ήχος
   Υπολογιστές
     Περιφερειακά
     Laptops
 Σπίτι
   Γραφείο
   Κουζίνα
(13 rows)
```

Προς τα πάνω

Η αναδρομή δουλεύει και αντίστροφα. Το breadcrumb μιας κατηγορίας, η αλυσίδα των γονέων της μέχρι τη ρίζα, ξεκινά από την κατηγορία και σε κάθε βήμα βρίσκει τη γονική της.

SQL

```
WITH RECURSIVE up AS (
  SELECT id, name, parent_id, 0 AS level
  FROM categories
  WHERE name = 'Laptops'
  UNION ALL
  SELECT c.id, c.name, c.parent_id, u.level + 1
  FROM categories AS c
  JOIN up AS u ON c.id = u.parent_id
)
SELECT string_agg(name, ' / ' ORDER BY level DESC) AS breadcrumb
FROM up;
```

ΑΠΟΤΕΛΕΣΜΑ

```
             breadcrumb
-----------------------------------
 Ηλεκτρονικά / Υπολογιστές / Laptops
(1 row)
```

Η μόνη διαφορά από τη διάσχιση προς τα κάτω είναι η κατεύθυνση του join, `c.id = u.parent_id` αντί για `c.parent_id = t.id`.

Aggregates πάνω σε υποδέντρα

Ένα συχνό πρόβλημα είναι να αθροίσεις κάτι για μια κατηγορία μαζί με όλες τις υποκατηγορίες της. Η λύση είναι να κρατάς σε κάθε γραμμή της αναδρομής από ποια ρίζα ξεκίνησε.

```sql
WITH RECURSIVE tree AS (
  SELECT id AS root_id, id
  FROM categories
  WHERE parent_id IS NULL
  UNION ALL
  SELECT t.root_id, c.id
  FROM categories AS c
  JOIN tree AS t ON c.parent_id = t.id
)
SELECT root.name AS root_category, count(p.id) AS products
FROM tree AS t
JOIN categories AS root ON root.id = t.root_id
LEFT JOIN products AS p ON p.category_id = t.id AND p.active
GROUP BY root.name
ORDER BY products DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 root_category | products
---------------+----------
 Ηλεκτρονικά   |       12
 Βιβλία        |       10
 Σπίτι         |        6
(3 rows)
```

Κάθε γραμμή του `tree` είναι ένα ζεύγος ρίζας και απογόνου. Μετά το join με τα προϊόντα, η ομαδοποίηση ανά ρίζα μετρά τα προϊόντα όλου του υποδέντρου.

Γράφοι και κύκλοι

Σε ένα δέντρο η αναδρομή σταματά πάντα, αφού κάθε διαδρομή καταλήγει σε φύλλο. Σε έναν γράφο με κύκλους μπορεί να επιστρέφει στα ίδια σημεία για πάντα. Ας ορίσουμε τις διαδρομές διανομής ανάμεσα στις πόλεις, με διάρκεια σε ώρες.

```sql
CREATE TABLE routes (
  from_city integer REFERENCES cities (id),
  to_city   integer REFERENCES cities (id),
  hours     numeric(4,1) NOT NULL,
  PRIMARY KEY (from_city, to_city)
);

INSERT INTO routes VALUES
  (1, 3, 3.0), (3, 1, 3.0), (1, 5, 4.0), (5, 6, 1.0), (6, 5, 1.0),
  (5, 2, 2.5), (2, 5, 2.5), (1, 4, 9.0), (4, 8, 2.0), (8, 4, 2.0),
  (3, 7, 3.5), (7, 2, 3.0), (1, 9, 3.0), (1, 10, 12.0);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 14
```

Οι πόλεις που φτάνεις από την Αθήνα, με τον συνολικό χρόνο, βρίσκονται με αναδρομή που προσθέτει μια ακμή σε κάθε βήμα. Η πρώτη προσπάθεια αποτυγχάνει για έναν λόγο που δεν έχει σχέση με τους κύκλους.

SQL

```
WITH RECURSIVE reach AS (
  SELECT r.to_city AS city, r.hours AS total_hours
  FROM routes AS r
  WHERE r.from_city = 1
  UNION ALL
  SELECT r.to_city, re.total_hours + r.hours
  FROM routes AS r
  JOIN reach AS re ON r.from_city = re.city
)
SELECT * FROM reach;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  recursive query "reach" column 2 has type numeric(4,1) in non-recursive term but type numeric overall
LINE 2:   SELECT r.to_city AS city, r.hours AS total_hours
                                    ^
HINT:  Cast the output of the non-recursive term to the correct type.
```

Οι στήλες του anchor ορίζουν τους τύπους του CTE. Το `hours` είναι `numeric(4,1)`, ενώ το άθροισμα στον αναδρομικό όρο είναι σκέτο `numeric`. Η PostgreSQL απαιτεί ακριβώς ίδιους τύπους και το hint λέει τη λύση. Μετατρέπεις τη στήλη του anchor.

Υπάρχει και δεύτερο πρόβλημα. Ο γράφος έχει κύκλους, για παράδειγμα Λάρισα, Βόλος και ξανά Λάρισα, οπότε η αναδρομή δεν θα σταματούσε ποτέ. Για να μη γυρίζει σε πόλεις που έχει ήδη περάσει, η PostgreSQL έχει το `CYCLE`. Σημαδεύει τη γραμμή που ξαναβρίσκει πόλη του μονοπατιού και σταματά εκεί.

SQL

```
WITH RECURSIVE reach AS (
  SELECT r.to_city AS city, r.hours::numeric AS total_hours, 1 AS legs
  FROM routes AS r
  WHERE r.from_city = 1
  UNION ALL
  SELECT r.to_city, re.total_hours + r.hours, re.legs + 1
  FROM routes AS r
  JOIN reach AS re ON r.from_city = re.city
) CYCLE city SET is_cycle USING path
SELECT DISTINCT ON (c.name) c.name, re.total_hours, re.legs
FROM reach AS re
JOIN cities AS c ON c.id = re.city
WHERE NOT re.is_cycle AND re.city <> 1
ORDER BY c.name, re.total_hours;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    name     | total_hours | legs
-------------+-------------+------
 Βόλος       |         5.0 |    2
 Ηράκλειο    |         9.0 |    1
 Θεσσαλονίκη |         6.5 |    2
 Ιωάννινα    |         6.5 |    2
 Καλαμάτα    |         3.0 |    1
 Λάρισα      |         4.0 |    1
 Πάτρα       |         3.0 |    1
 Ρόδος       |        12.0 |    1
 Χανιά       |        11.0 |    2
(9 rows)
```

Το `CYCLE city SET is_cycle USING path` προσθέτει δύο στήλες. Η `path` κρατά τις πόλεις που πέρασε κάθε μονοπάτι και η `is_cycle` γίνεται αληθής όταν η νέα πόλη υπάρχει ήδη στο μονοπάτι. Οι γραμμές με `is_cycle` αληθές δεν συνεχίζουν. Το `DISTINCT ON` κρατά τη συντομότερη διαδρομή για κάθε πόλη.

ΠΑΓΙΔΑ

Μια αναδρομή χωρίς συνθήκη τερματισμού σε γράφο με κύκλο δεν τελειώνει ποτέ. Όταν γράφεις ένα recursive CTE, ρώτα πρώτα γιατί θα σταματήσει. Σε δέντρα σταματά στα φύλλα. Σε γράφους χρειάζεσαι `CYCLE` ή ένα όριο βάθους, όπως `WHERE legs < 10`.

ΚΡΑΤΑ ΑΥΤΟ

Το recursive CTE έχει anchor και αναδρομικό όρο ενωμένα με `UNION ALL`. Ο αναδρομικός όρος βλέπει μόνο τις γραμμές του προηγούμενου βήματος και η αναδρομή σταματά όταν δεν παράγει νέες. Για σειρά δέντρου χρησιμοποίησε μονοπάτι ή `SEARCH`. Για γράφους με κύκλους χρησιμοποίησε `CYCLE`.

Ασκήσεις

ΑΣΚΗΣΗ 22.1 Το υποδέντρο των ηλεκτρονικών

Εμφάνισε όλες τις κατηγορίες κάτω από τα Ηλεκτρονικά, μαζί με αυτήν, με το βάθος τους και ταξινομημένες κατά μονοπάτι.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |     name      | depth |                   path
----+---------------+-------+-------------------------------------------
  6 | Ηλεκτρονικά   |     0 | Ηλεκτρονικά
 10 | Ήχος          |     1 | Ηλεκτρονικά / Ήχος
  7 | Υπολογιστές   |     1 | Ηλεκτρονικά / Υπολογιστές
  9 | Περιφερειακά  |     2 | Ηλεκτρονικά / Υπολογιστές / Περιφερειακά
  8 | Laptops       |     2 | Ηλεκτρονικά / Υπολογιστές / Laptops
(5 rows)
```

ΑΣΚΗΣΗ 22.2 Breadcrumb κάθε προϊόντος

Για κάθε ενεργό προϊόν της κατηγορίας 4 και της κατηγορίας 9 εμφάνισε το όνομά του και το πλήρες μονοπάτι της κατηγορίας του από τη ρίζα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             |                    path
-----------------------------+-----------------------------------------
 Καθαρός κώδικας στην πράξη | Βιβλία / Τεχνολογία / Προγραμματισμός
 Μαθαίνω Python              | Βιβλία / Τεχνολογία / Προγραμματισμός
 Ruby από την αρχή           | Βιβλία / Τεχνολογία / Προγραμματισμός
 Ασύρματο ποντίκι            | Ηλεκτρονικά / Υπολογιστές / Περιφερειακά
 Μηχανικό πληκτρολόγιο       | Ηλεκτρονικά / Υπολογιστές / Περιφερειακά
 Οθόνη 27" 4K                | Ηλεκτρονικά / Υπολογιστές / Περιφερειακά
 USB-C docking station       | Ηλεκτρονικά / Υπολογιστές / Περιφερειακά
 Webcam HD                   | Ηλεκτρονικά / Υπολογιστές / Περιφερειακά
(8 rows)
```

ΑΣΚΗΣΗ 22.3 Η ομάδα των Πωλήσεων

Βρες όλους τους υπαλλήλους που αναφέρονται άμεσα ή έμμεσα στον υπάλληλο 3. Εμφάνισε όνομα, θέση και επίπεδο, όπου 1 σημαίνει άμεση αναφορά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    full_name    |      title      | level
-----------------+-----------------+-------
 Ιωάννα Ρήγα     | Account Manager |     1
 Στέλιος Παππάς  | Account Manager |     1
 Βασιλική Ντόκου | Sales Intern    |     2
(3 rows)
```

ΑΣΚΗΣΗ 22.4 Πόσους έχει από κάτω του κάθε προϊστάμενος

Για κάθε υπάλληλο που έχει τουλάχιστον έναν υφιστάμενο, εμφάνισε το όνομά του, πόσους έχει άμεσα και πόσους συνολικά σε όλα τα επίπεδα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       full_name       | direct | total
-----------------------+--------+-------
 Ελένη Μαυρίδου        |      3 |    13
 Νίκος Αλεξίου         |      3 |     4
 Δημήτρης Λαμπρόπουλος |      2 |     3
 Σοφία Κωνσταντίνου    |      2 |     3
 Γιάννης Κορρές        |      1 |     1
 Ιωάννα Ρήγα           |      1 |     1
 Κατερίνα Ζαφειρίου    |      1 |     1
(7 rows)
```

ΑΣΚΗΣΗ 22.5 Έσοδα ανά κορυφαία κατηγορία

Υπολόγισε τα έσοδα από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν για κάθε μία από τις τρεις ρίζες του δέντρου, μαζί με όλες τις υποκατηγορίες τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 root_category | revenue
---------------+----------
 Ηλεκτρονικά   | 48232.10
 Σπίτι         | 12320.90
 Βιβλία        |  3680.10
(3 rows)
```

ΑΣΚΗΣΗ 22.6 Η συντομότερη διαδρομή

Με τον πίνακα `routes` της θεωρίας, βρες τη συντομότερη σε ώρες διαδρομή από την Αθήνα στη Θεσσαλονίκη. Εμφάνισε τις ώρες και τη διαδρομή με ονόματα πόλεων, όπως `Αθήνα / Λάρισα / Θεσσαλονίκη`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 total_hours |            route
-------------+-----------------------------
         6.5 | Αθήνα / Λάρισα / Θεσσαλονίκη
(1 row)
```

ΚΕΦΑΛΑΙΟ 23

Window functions για κατάταξη και top N

Μια window function υπολογίζει κάτι πάνω σε ένα σύνολο γραμμών, όπως ένα aggregate, αλλά δεν τις συμπτύσσει σε μία. Κάθε γραμμή μένει στο αποτέλεσμα και παίρνει δίπλα της την τιμή που της αντιστοιχεί. Έτσι γράφονται κατατάξεις, ποσοστά επί του συνόλου και top N ανά ομάδα.

Aggregate χωρίς σύμπτυξη

Θέλεις κάθε προϊόν της κατηγορίας 9 με την τιμή του και δίπλα τη μέση τιμή της κατηγορίας. Με `GROUP BY` θα έχανες τα προϊόντα και θα έμενε μία γραμμή. Με `OVER` το aggregate υπολογίζεται χωρίς να μαζέψει τις γραμμές.

SQL

```
SELECT name, price,
       round(avg(price) OVER (), 2) AS avg_price,
       round(100 * price / sum(price) OVER (), 1) AS pct_of_total
FROM products
WHERE category_id = 9
ORDER BY price DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name          | price  | avg_price | pct_of_total
-----------------------+--------+-----------+--------------
 Οθόνη 27" 4K          | 329.00 |    128.18 |         51.3
 USB-C docking station | 139.00 |    128.18 |         21.7
 Μηχανικό πληκτρολόγιο |  89.00 |    128.18 |         13.9
 Webcam HD             |  59.00 |    128.18 |          9.2
 Ασύρματο ποντίκι      |  24.90 |    128.18 |          3.9
(5 rows)
```

Το `OVER ()` λέει ότι το `avg` είναι window function και ότι το «παράθυρο» κάθε γραμμής είναι όλες οι γραμμές του αποτελέσματος. Κάθε γραμμή παίρνει τον ίδιο μέσο όρο και το ίδιο άθροισμα, οπότε το ποσοστό της υπολογίζεται στην ίδια έκφραση.

PARTITION BY

Το `PARTITION BY` χωρίζει τις γραμμές σε ομάδες όπως το `GROUP BY`. Το παράθυρο κάθε γραμμής είναι η ομάδα της. Η διαφορά είναι ότι οι γραμμές δεν συμπτύσσονται.

SQL

```
SELECT category_id, name, price,
       round(avg(price) OVER (PARTITION BY category_id), 2) AS category_avg,
       count(*) OVER (PARTITION BY category_id) AS in_category
FROM products
WHERE category_id IN (8, 10)
ORDER BY category_id, price DESC;
```

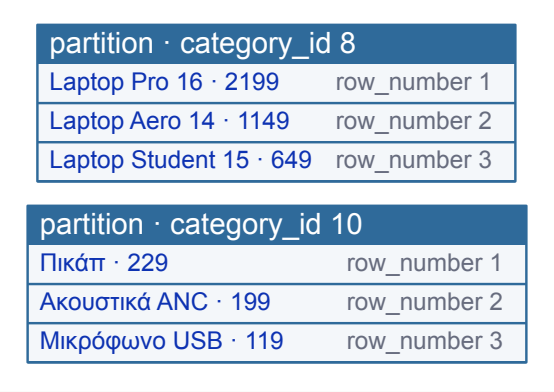
ΑΠΟΤΕΛΕΣΜΑ

```
 category_id |       name        |  price  | category_avg | in_category
-------------+-------------------+---------+--------------+-------------
           8 | Laptop Pro 16     | 2199.00 |      1199.00 |           4
           8 | Laptop Aero 14    | 1149.00 |      1199.00 |           4
           8 | Laptop Classic 13 |  799.00 |      1199.00 |           4
           8 | Laptop Student 15 |  649.00 |      1199.00 |           4
          10 | Πικάπ             |  229.00 |       156.50 |           4
          10 | Ακουστικά ANC     |  199.00 |       156.50 |           4
          10 | Μικρόφωνο USB     |  119.00 |       156.50 |           4
          10 | Ηχείο Bluetooth   |   79.00 |       156.50 |           4
(8 rows)
```

Αυτό το query στο κεφάλαιο 14 χρειαζόταν correlated subquery για κάθε γραμμή. Εδώ είναι μία έκφραση και η βάση υπολογίζει κάθε partition μία φορά.

Συναρτήσεις κατάταξης

Με `ORDER BY` μέσα στο `OVER` οι γραμμές κάθε partition έχουν σειρά και μπορούν να αριθμηθούν. Υπάρχουν τρεις συναρτήσεις κατάταξης που διαφέρουν μόνο στις ισοβαθμίες.



Σχήμα 23.1 · Το PARTITION BY χωρίζει τις γραμμές σε ομάδες και το ORDER BY τις ταξινομεί μέσα σε καθεμία. Κάθε γραμμή μένει στο αποτέλεσμα με τη θέση της.

SQL

```
SELECT name, stock,
       row_number() OVER (ORDER BY stock) AS row_number,
       rank() OVER (ORDER BY stock) AS rank,
       dense_rank() OVER (ORDER BY stock) AS dense_rank
FROM products
WHERE stock < 10
ORDER BY stock, name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name             | stock | row_number | rank | dense_rank
-----------------------------+-------+------------+------+------------
 Ματωμένα χώματα             |     0 |          1 |    1 |          1
 Laptop Classic 13           |     0 |          3 |    1 |          1
 Webcam HD                   |     0 |          2 |    1 |          1
 Γραφείο ρυθμιζόμενου ύψους  |     3 |          5 |    4 |          2
 Πικάπ                       |     3 |          4 |    4 |          2
 Laptop Pro 16               |     4 |          6 |    6 |          3
 Δίκτυα υπολογιστών          |     6 |          8 |    7 |          4
 Εργονομική καρέκλα          |     6 |          7 |    7 |          4
 Σχεδίαση βάσεων δεδομένων   |     7 |          9 |    9 |          5
 Laptop Aero 14              |     8 |         10 |   10 |          6
 Καφετιέρα espresso          |     9 |         11 |   11 |          7
 Ruby από την αρχή           |     9 |         12 |   11 |          7
(12 rows)
```

Το `row_number` δίνει διαδοχικούς αριθμούς χωρίς επαναλήψεις. Στις ισοβαθμίες η σειρά είναι αυθαίρετη. Το `rank` δίνει τον ίδιο αριθμό στις ισοβαθμίες και μετά πηδά, όπως στους αγώνες. Τρία προϊόντα μοιράζονται την πρώτη θέση και το επόμενο είναι τέταρτο. Το `dense_rank` δεν πηδά. Το επόμενο είναι δεύτερο.

ΠΑΓΙΔΑ

Αν χρησιμοποιείς `row_number` για να διαλέξεις «την πρώτη» γραμμή, βάλε στο `ORDER BY` αρκετά κλειδιά ώστε να μην υπάρχουν ισοβαθμίες, όπως στο κεφάλαιο 4. Αλλιώς ποια γραμμή θα πάρει τον αριθμό 1 μπορεί να αλλάζει από εκτέλεση σε εκτέλεση.

Top N ανά ομάδα

Οι window functions υπολογίζονται μετά το `WHERE`, το `GROUP BY` και το `HAVING`, αλλά πριν από το `ORDER BY` και το `LIMIT`. Γι' αυτό δεν μπορείς να φιλτράρεις με το αποτέλεσμά τους στο `WHERE` του ίδιου query.

SQL

```
SELECT name, category_id, price,
       row_number() OVER (PARTITION BY category_id ORDER BY price DESC) AS pos
FROM products
WHERE row_number() OVER (PARTITION BY category_id ORDER BY price DESC) <= 2;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  window functions are not allowed in WHERE
LINE 4: WHERE row_number() OVER (PARTITION BY category_id ORDER BY p...
              ^
```

Υπολογίζεις τη θέση σε CTE ή derived table και φιλτράρεις στο εξωτερικό query. Αυτό είναι το πιο συνηθισμένο μοτίβο με window functions.

SQL

```
WITH ranked AS (
  SELECT name, category_id, price,
         row_number() OVER (PARTITION BY category_id ORDER BY price DESC, id) AS pos
  FROM products
  WHERE active AND category_id IN (8, 9, 10)
)
SELECT category_id, pos, name, price
FROM ranked
WHERE pos <= 2
ORDER BY category_id, pos;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 category_id | pos |         name          |  price
-------------+-----+-----------------------+---------
           8 |   1 | Laptop Pro 16         | 2199.00
           8 |   2 | Laptop Aero 14        | 1149.00
           9 |   1 | Οθόνη 27" 4K          |  329.00
           9 |   2 | USB-C docking station |  139.00
          10 |   1 | Πικάπ                 |  229.00
          10 |   2 | Ακουστικά ANC         |  199.00
(6 rows)
```

Είναι το ίδιο αποτέλεσμα με το `LATERAL` της άσκησης 14.5. Το `LATERAL` είναι συνήθως ταχύτερο όταν υπάρχει κατάλληλο index και οι ομάδες είναι λίγες. Το `row_number` διαβάζει όλες τις γραμμές μία φορά και είναι απλούστερο όταν οι ομάδες είναι πολλές. Το κεφάλαιο 36 συγκρίνει τα δύο.

Η πιο πρόσφατη παραγγελία κάθε πελάτη, το πρόβλημα των κεφαλαίων 4 και 14, γράφεται με τον ίδιο τρόπο. Είναι ο τέταρτος τρόπος και ο πιο φορητός, αφού οι window functions υπάρχουν σε όλες τις σύγχρονες βάσεις.

SQL

```
SELECT customer_id, id AS order_id, created_at::date
FROM (SELECT customer_id, id, created_at,
             row_number() OVER (PARTITION BY customer_id ORDER BY created_at DESC) AS rn
      FROM orders) AS t
WHERE rn = 1 AND customer_id <= 4
ORDER BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | order_id | created_at
-------------+----------+------------
           1 |      129 | 2026-05-26
           2 |      128 | 2026-05-25
           3 |      145 | 2026-06-14
           4 |      141 | 2026-06-09
(4 rows)
```

Window functions πάνω σε aggregates

Ένα query με `GROUP BY` μπορεί να έχει και window functions. Υπολογίζονται πάνω στις ομάδες, αφού αυτές είναι οι γραμμές του αποτελέσματος τη στιγμή που εκτελούνται.

SQL

```
SELECT cat.name AS category,
       sum(oi.quantity * oi.unit_price) AS revenue,
       rank() OVER (ORDER BY sum(oi.quantity * oi.unit_price) DESC) AS position,
       round(100 * sum(oi.quantity * oi.unit_price)
             / sum(sum(oi.quantity * oi.unit_price)) OVER (), 1) AS pct
FROM order_items AS oi
JOIN orders AS o ON o.id = oi.order_id
JOIN products AS p ON p.id = oi.product_id
JOIN categories AS cat ON cat.id = p.category_id
WHERE o.status NOT IN ('cancelled', 'refunded')
GROUP BY cat.name
ORDER BY position;
```

ΑΠΟΤΕΛΕΣΜΑ

```
     category     | revenue  | position | pct
------------------+----------+----------+------
 Laptops          | 36572.00 |        1 | 56.9
 Γραφείο          |  8800.00 |        2 | 13.7
 Περιφερειακά     |  6675.10 |        3 | 10.4
 Ήχος             |  4985.00 |        4 |  7.8
 Κουζίνα          |  3520.90 |        5 |  5.5
 Προγραμματισμός  |  1292.00 |        6 |  2.0
 Βάσεις δεδομένων |  1125.00 |        7 |  1.8
 Λογοτεχνία       |   771.10 |        8 |  1.2
 Τεχνολογία       |   492.00 |        9 |  0.8
(9 rows)
```

Το `sum(sum(...)) OVER ()` φαίνεται παράξενο αλλά διαβάζεται εύκολα από μέσα προς τα έξω. Το εσωτερικό `sum` είναι το aggregate του `GROUP BY` και δίνει τα έσοδα κάθε κατηγορίας. Το εξωτερικό είναι window function και αθροίζει τα έσοδα όλων των κατηγοριών.

NTILE και ονομασμένα παράθυρα

Το `ntile(n)` μοιράζει τις γραμμές κάθε partition σε `n` ομάδες όσο γίνεται ισομεγέθεις. Είναι ο γρήγορος τρόπος για τεταρτημόρια ή δεκατημόρια.

Όταν πολλές window functions χρησιμοποιούν το ίδιο παράθυρο, το ορίζεις μία φορά στο `WINDOW` και το αναφέρεις με όνομα.

SQL

```
SELECT customer_id, spent,
       ntile(4) OVER w AS quartile,
       rank() OVER w AS position
FROM (SELECT o.customer_id, sum(oi.quantity * oi.unit_price) AS spent
      FROM orders AS o
      JOIN order_items AS oi ON oi.order_id = o.id
      WHERE o.status NOT IN ('cancelled', 'refunded')
      GROUP BY o.customer_id) AS s
WINDOW w AS (ORDER BY spent DESC)
ORDER BY position
LIMIT 8;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id |  spent   | quartile | position
-------------+----------+----------+----------
           9 | 7946.20 |        1 |        1
          23 | 7291.60 |        1 |        2
           2 | 6630.00 |        1 |        3
          14 | 5177.50 |        1 |        4
           8 | 4401.80 |        1 |        5
          13 | 3786.20 |        1 |        6
           4 | 3449.90 |        1 |        7
           5 | 2816.50 |        2 |        8
(8 rows)
```

ΚΡΑΤΑ ΑΥΤΟ

Το `OVER` κάνει ένα aggregate ή μια συνάρτηση κατάταξης window function. Το `PARTITION BY` ορίζει τις ομάδες και το `ORDER BY` τη σειρά μέσα σε αυτές. Οι γραμμές δεν συμπτύσσονται. Οι window functions υπολογίζονται μετά το `HAVING`, οπότε για top N τις υπολογίζεις σε CTE και φιλτράρεις έξω.

Ασκήσεις

ΑΣΚΗΣΗ 23.1 Απόσταση από τον μέσο όρο

Για τα ενεργά προϊόντα των κατηγοριών 12 και 13 εμφάνισε όνομα, τιμή, μέση τιμή της κατηγορίας τους και τη διαφορά της τιμής από αυτήν, με δύο δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 category_id |            name             | price  | category_avg |  diff
-------------+-----------------------------+--------+--------------+---------
          12 | Καφετιέρα espresso          | 249.00 |       115.97 |  133.03
          12 | Σετ μαχαιριών               |  64.00 |       115.97 |  -51.97
          12 | Βραστήρας                   |  34.90 |       115.97 |  -81.07
          13 | Γραφείο ρυθμιζόμενου ύψους | 459.00 |       263.33 |  195.67
          13 | Εργονομική καρέκλα          | 289.00 |       263.33 |   25.67
          13 | Φωτιστικό γραφείου LED      |  42.00 |       263.33 | -221.33
(6 rows)
```

ΑΣΚΗΣΗ 23.2 Κατάταξη μισθών ανά τμήμα

Για κάθε υπάλληλο με τμήμα και μισθό εμφάνισε τμήμα, όνομα, μισθό και τη θέση του μισθού του μέσα στο τμήμα, από τον υψηλότερο, με `dense_rank`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 department |       full_name       | salary  | position
------------+-----------------------+---------+----------
 Αποθήκη    | Δημήτρης Λαμπρόπουλος | 6200.00 |        1
 Αποθήκη    | Γιάννης Κορρές        | 1800.00 |        2
 Αποθήκη    | Θανάσης Μπέης         | 1750.00 |        3
 Αποθήκη    | Χριστίνα Αθανασίου    | 1650.00 |        4
 Διοίκηση   | Ελένη Μαυρίδου        | 9500.00 |        1
 Πωλήσεις   | Σοφία Κωνσταντίνου    | 7000.00 |        1
 Πωλήσεις   | Ιωάννα Ρήγα           | 3100.00 |        2
 Πωλήσεις   | Στέλιος Παππάς        | 2900.00 |        3
 Πωλήσεις   | Βασιλική Ντόκου       | 1100.00 |        4
 Τεχνολογία | Νίκος Αλεξίου         | 8200.00 |        1
 Τεχνολογία | Μαρίνα Σταθοπούλου    | 4400.00 |        2
 Τεχνολογία | Κατερίνα Ζαφειρίου    | 4200.00 |        3
 Τεχνολογία | Άγγελος Φωτίου        | 3900.00 |        4
 Τεχνολογία | Πέτρος Χατζής         | 2100.00 |        5
(14 rows)
```

ΑΣΚΗΣΗ 23.3 Οι δύο καλύτεροι πελάτες κάθε περιφέρειας

Για κάθε περιφέρεια εμφάνισε τους δύο πελάτες με τα περισσότερα έσοδα από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν. Εμφάνισε περιφέρεια, θέση, όνομα και έσοδα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       region       | pos | first_name |  last_name   | revenue
--------------------+-----+------------+--------------+---------
 Αττική             |   1 | Στέλιος    | Κουτσούκος   | 7291.60
 Αττική             |   2 | Παναγιώτης | Βασιλείου    | 4401.80
 Δυτική Ελλάδα      |   1 | Κώστας     | Δημητρίου    | 3449.90
 Δυτική Ελλάδα      |   2 | Θανάσης    | Καραγιάννης  | 1642.70
 Ήπειρος            |   1 | Ιωάννα     | Παππά        | 2101.80
 Θεσσαλία           |   1 | Σοφία      | Αλεξίου      | 7946.20
 Θεσσαλία           |   2 | Δήμητρα    | Κωνσταντίνου | 1101.10
 Κεντρική Μακεδονία |   1 | Γιώργος    | Παπαδόπουλος | 6630.00
 Κεντρική Μακεδονία |   2 | Βασίλης    | Αντωνίου     | 5177.50
 Κρήτη              |   1 | Κατερίνα   | Σταύρου      | 3786.20
 Κρήτη              |   2 | Αναστασία  | Γεωργίου     | 2816.50
 Νότιο Αιγαίο       |   1 | Σπύρος     | Μαρκόπουλος  |  662.50
 Πελοπόννησος       |   1 | Ευαγγελία  | Χριστοδούλου | 1158.60
(13 rows)
```

ΑΣΚΗΣΗ 23.4 Μερίδιο κάθε παραγγελίας

Για τις παραγγελίες του πελάτη 13 εμφάνισε `id`, αξία και το ποσοστό της αξίας στο σύνολο των παραγγελιών του πελάτη, με ένα δεκαδικό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 order_id |  total  | pct
----------+---------+------
       20 |  114.90 |  3.0
       35 |  690.80 | 18.2
       40 | 2527.50 | 66.8
       41 |  289.00 |  7.6
       82 |  164.00 |  4.3
(5 rows)
```

ΑΣΚΗΣΗ 23.5 Διπλές κριτικές με row_number

Βρες με `row_number` τις κριτικές που δεν είναι η πιο πρόσφατη για το ζεύγος προϊόντος και πελάτη τους. Εμφάνισε `id`, `product_id`, `customer_id` και `created_at`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | product_id | customer_id |       created_at
----+------------+-------------+------------------------
 25 |         11 |           2 | 2025-07-18 10:03:00+03
(1 row)
```

ΑΣΚΗΣΗ 23.6 Τεταρτημόρια προϊόντων

Χώρισε τα ενεργά προϊόντα με κατηγορία σε τέσσερις ομάδες κατά τιμή, με την 1 για τα ακριβότερα. Εμφάνισε για κάθε ομάδα το πλήθος, τη μικρότερη και τη μεγαλύτερη τιμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 quartile | products | min_price | max_price
----------+----------+-----------+-----------
        1 |        7 |    249.00 |   2199.00
        2 |        7 |     64.00 |    229.00
        3 |        7 |     34.90 |     59.00
        4 |        7 |      9.90 |     32.00
(4 rows)
```

ΑΣΚΗΣΗ 23.7 Η τρίτη παραγγελία κάθε πελάτη

Για τους πελάτες 1 έως 6 εμφάνισε την τρίτη χρονολογικά παραγγελία τους με `id` και ημερομηνία.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | id | created_at
-------------+----+------------
           1 | 12 | 2025-03-25
           2 | 23 | 2025-05-15
           3 | 32 | 2025-07-29
           4 | 61 | 2025-11-26
           5 | 51 | 2025-10-27
           6 | 26 | 2025-06-18
(6 rows)
```

ΚΕΦΑΛΑΙΟ 24

Window functions με frames, LAG και LEAD

Όταν ένα παράθυρο έχει `ORDER BY`, κάθε γραμμή βλέπει μόνο ένα κομμάτι του, το frame. Με αυτό γράφονται τρέχοντα σύνολα και κινούμενοι μέσοι όροι. Το `LAG` και το `LEAD` διαβάζουν την προηγούμενη και την επόμενη γραμμή, ώστε να συγκρίνεις κάθε γραμμή με τις γειτονικές της.

Τρέχον σύνολο

Στο προηγούμενο κεφάλαιο το `sum(...) OVER ()` έδινε σε κάθε γραμμή το ίδιο άθροισμα όλων. Αν προσθέσεις `ORDER BY` στο παράθυρο, κάθε γραμμή αθροίζει μόνο τις γραμμές μέχρι τον εαυτό της. Αυτό είναι το τρέχον σύνολο.

```sql
WITH daily AS (
  SELECT o.created_at::date AS day, sum(oi.quantity * oi.unit_price) AS revenue
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  WHERE o.status NOT IN ('cancelled', 'refunded')
    AND o.created_at >= '2026-06-01'
  GROUP BY 1
)
SELECT day, revenue,
       sum(revenue) OVER (ORDER BY day) AS running_total
FROM daily
ORDER BY day;
```

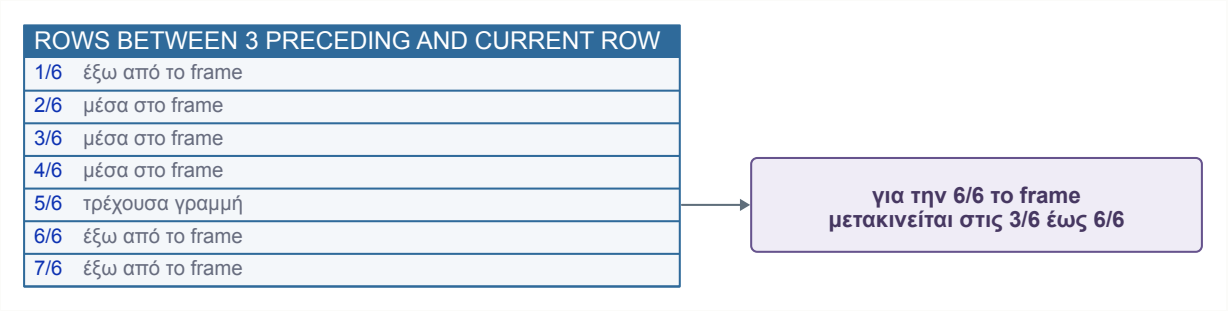
ΑΠΟΤΕΛΕΣΜΑ

```
    day     | revenue | running_total
------------+---------+---------------
 2026-06-03 |   42.00 |         42.00
 2026-06-04 |  867.00 |        909.00
 2026-06-05 |  649.00 |       1558.00
 2026-06-06 |  185.00 |       1743.00
 2026-06-08 |   80.00 |       1823.00
 2026-06-09 | 2353.50 |       4176.50
 2026-06-13 |  367.90 |       4544.40
 2026-06-14 |  725.90 |       5270.30
 2026-06-15 |  530.60 |       5800.90
 2026-06-17 |  228.00 |       6028.90
 2026-06-19 |   46.50 |       6075.40
 2026-06-21 |  688.80 |       6764.20
 2026-06-23 |  751.40 |       7515.60
 2026-06-24 | 1316.00 |       8831.60
 …
(16 rows)
```

Η τελευταία τιμή του `running_total` είναι τα έσοδα όλου του Ιουνίου. Κάθε ενδιάμεση τιμή είναι τα έσοδα από την αρχή του μήνα μέχρι εκείνη την ημέρα.

Frames

Το κομμάτι του παραθύρου που βλέπει κάθε γραμμή λέγεται **frame**. Όταν το `OVER` έχει `ORDER BY` χωρίς ρητό frame, η προεπιλογή είναι `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Από την αρχή του partition μέχρι την τρέχουσα γραμμή. Μπορείς να γράψεις δικό σου.



Σχήμα 24.1 · Το frame ορίζεται σε σχέση με την τρέχουσα γραμμή. Κάθε γραμμή έχει το δικό της frame, που μετακινείται μαζί της.

Υπάρχουν τρεις τρόποι να μετρήσεις τα όρια του frame. Το `ROWS` μετρά γραμμές. Το `RANGE` μετρά τιμές του κλειδιού ταξινόμησης. Το `GROUPS` μετρά ομάδες γραμμών με την ίδια τιμή. Η διαφορά ανάμεσα στο `ROWS` και στο `RANGE` φαίνεται όταν υπάρχουν ισοβαθμίες.

SQL

```
SELECT id, created_at::date AS day,
       count(*) OVER (ORDER BY created_at::date) AS range_default,
       count(*) OVER (ORDER BY created_at::date
                      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS rows_frame
FROM orders
WHERE created_at >= '2026-06-13' AND created_at < '2026-06-16'
ORDER BY created_at, id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id  |    day     | range_default | rows_frame
-----+------------+---------------+------------
 142 | 2026-06-13 |             2 |          1
 143 | 2026-06-13 |             2 |          2
 144 | 2026-06-14 |             6 |          3
 145 | 2026-06-14 |             6 |          4
 146 | 2026-06-14 |             6 |          5
 147 | 2026-06-14 |             6 |          6
 148 | 2026-06-15 |            10 |          7
 149 | 2026-06-15 |            10 |          8
 150 | 2026-06-15 |            10 |          9
 151 | 2026-06-15 |            10 |         10
(10 rows)
```

Με την προεπιλογή `RANGE`, όλες οι παραγγελίες της ίδιας ημέρας είναι **peers**, δηλαδή ισότιμες γραμμές. Μπαίνουν όλες μαζί στο frame. Οι τέσσερις παραγγελίες της 14ης βλέπουν όλες το 6. Με `ROWS` κάθε γραμμή μετρά μέχρι τον εαυτό της και ο αριθμός αυξάνεται κατά ένα.

ΠΑΓΙΔΑ

Ένα τρέχον σύνολο με `ORDER BY` σε στήλη με ισοβαθμίες δίνει το ίδιο σύνολο σε όλες τις ισότιμες γραμμές. Αν θέλεις γραμμή προς γραμμή, γράψε `ROWS` και βάλε στο `ORDER BY` ένα μοναδικό κλειδί.

Κινούμενος μέσος όρος

Ένα frame με όρια και από τις δύο πλευρές κινείται μαζί με τη γραμμή. Ο κινούμενος μέσος όρος τριών ημερών με παραγγελίες είναι αυτός.

SQL

```
WITH daily AS (
  SELECT o.created_at::date AS day, sum(oi.quantity * oi.unit_price) AS revenue
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  WHERE o.status NOT IN ('cancelled', 'refunded')
    AND o.created_at >= '2026-06-01'
  GROUP BY 1
)
SELECT day, revenue,
       round(avg(revenue) OVER (ORDER BY day ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 2)
         AS avg_3_rows,
       round(avg(revenue) OVER (ORDER BY day RANGE BETWEEN interval '2 days' PRECEDING
                                AND CURRENT ROW), 2) AS avg_3_days
FROM daily
ORDER BY day
LIMIT 8;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    day     | revenue | avg_3_rows | avg_3_days
------------+---------+------------+------------
 2026-06-03 |   42.00 |      42.00 |      42.00
 2026-06-04 |  867.00 |     454.50 |     454.50
 2026-06-05 |  649.00 |     519.33 |     519.33
 2026-06-06 |  185.00 |     567.00 |     567.00
 2026-06-08 |   80.00 |     304.67 |     132.50
 2026-06-09 | 2353.50 |     872.83 |    1216.75
 2026-06-13 |  367.90 |     933.80 |     367.90
 2026-06-14 |  725.90 |    1149.10 |     546.90
(8 rows)
```

Οι δύο στήλες διαφέρουν. Το `ROWS BETWEEN 2 PRECEDING` παίρνει τις τρεις τελευταίες γραμμές, όποιες ημέρες κι αν είναι. Στις 13 Ιουνίου αυτό σημαίνει και τις 8 και 9 Ιουνίου. Το `RANGE BETWEEN interval '2 days' PRECEDING` παίρνει τις ημέρες από 11 έως 13 Ιουνίου, δηλαδή μόνο τη 13η, αφού οι άλλες δεν έχουν παραγγελίες. Για χρονοσειρές με κενά το `RANGE` με `interval` είναι σχεδόν πάντα αυτό που εννοείς. Το κεφάλαιο 25 δείχνει πώς γεμίζεις τα κενά με μηδενικά ώστε να δουλεύουν και τα δύο.

LAG και LEAD

Το `lag(έκφραση)` επιστρέφει την τιμή της έκφρασης στην προηγούμενη γραμμή του partition. Το `lead(έκφραση)` στην επόμενη. Στην πρώτη γραμμή το `lag` δίνει `NULL`, αφού δεν υπάρχει προηγούμενη. Με δεύτερο όρισμα κοιτάζεις πιο μακριά και με τρίτο δίνεις προεπιλογή αντί για `NULL`.

SQL

```
SELECT id, created_at::date AS day,
       lag(created_at::date) OVER w AS previous_day,
       created_at::date - lag(created_at::date) OVER w AS days_between,
       lead(id) OVER w AS next_order
FROM orders
WHERE customer_id = 13
WINDOW w AS (ORDER BY created_at)
ORDER BY created_at;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |    day     | previous_day | days_between | next_order
----+------------+--------------+--------------+------------
 20 | 2025-05-04 | ∅            |            ∅ |         35
 35 | 2025-09-03 | 2025-05-04   |          122 |         40
 40 | 2025-09-19 | 2025-09-03   |           16 |         41
 41 | 2025-09-21 | 2025-09-19   |            2 |         82
 82 | 2026-02-04 | 2025-09-21   |          136 |          ∅
(5 rows)
```

Η διαφορά από την προηγούμενη γραμμή είναι η πιο συνηθισμένη χρήση. Η μεταβολή των εσόδων από μήνα σε μήνα είναι ένα `lag` πάνω σε ένα `GROUP BY`.

SQL

```
SELECT date_trunc('month', o.created_at)::date AS month,
       sum(oi.quantity * oi.unit_price) AS revenue,
       round(100 * (sum(oi.quantity * oi.unit_price)
                    / lag(sum(oi.quantity * oi.unit_price)) OVER (ORDER BY date_trunc('month', o.created_at))
                    - 1), 1) AS change_pct
FROM orders AS o
JOIN order_items AS oi ON oi.order_id = o.id
WHERE o.status NOT IN ('cancelled', 'refunded')
  AND o.created_at >= '2026-01-01'
GROUP BY date_trunc('month', o.created_at)
ORDER BY 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
   month    | revenue  | change_pct
------------+----------+------------
 2026-01-01 |  1860.80 |          ∅
 2026-02-01 |  3747.20 |      101.4
 2026-03-01 |  1599.20 |      -57.3
 2026-04-01 |  4026.50 |      151.8
 2026-05-01 | 10338.80 |      156.8
 2026-06-01 |  9074.90 |      -12.2
(6 rows)
```

Το query είναι σωστό αλλά δύσκολο να διαβαστεί, γιατί το άθροισμα επαναλαμβάνεται. Με ένα CTE για τα μηνιαία έσοδα και `lag` στο κύριο query γίνεται πολύ καθαρότερο. Αυτή είναι η άσκηση 24.3.

FIRST_VALUE, LAST_VALUE και η παγίδα του frame

Το `first_value` δίνει την τιμή της πρώτης γραμμής του frame και το `last_value` της τελευταίας. Το πρώτο δουλεύει όπως περιμένεις. Το δεύτερο όχι.

SQL

```
SELECT id, created_at::date AS day,
       first_value(id) OVER w AS first_order,
       last_value(id) OVER w AS last_order_wrong
FROM orders
WHERE customer_id = 13
WINDOW w AS (ORDER BY created_at)
ORDER BY created_at;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |    day     | first_order | last_order_wrong
----+------------+-------------+------------------
 20 | 2025-05-04 |          20 |               20
 35 | 2025-09-03 |          20 |               35
 40 | 2025-09-19 |          20 |               40
 41 | 2025-09-21 |          20 |               41
 82 | 2026-02-04 |          20 |               82
(5 rows)
```

Η στήλη `last_order_wrong` είναι απλώς το `id` κάθε γραμμής. Το frame από προεπιλογή τελειώνει στην τρέχουσα γραμμή, οπότε η τελευταία γραμμή του frame είναι η ίδια. Για την τελευταία γραμμή του partition πρέπει το frame να φτάνει ως το τέλος.

SQL

```
SELECT id, created_at::date AS day,
       last_value(id) OVER (ORDER BY created_at
                            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_order
FROM orders
WHERE customer_id = 13
ORDER BY created_at;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |    day     | last_order
----+------------+------------
 20 | 2025-05-04 |         82
 35 | 2025-09-03 |         82
 40 | 2025-09-19 |         82
 41 | 2025-09-21 |         82
 82 | 2026-02-04 |         82
(5 rows)
```

Το ίδιο γράφεται απλούστερα με `first_value` και αντίστροφη ταξινόμηση, που δεν εξαρτάται από το frame.

Σχετική θέση

Το `percent_rank()` δίνει τη σχετική θέση μιας γραμμής από 0 έως 1. Το `cume_dist()` δίνει το ποσοστό των γραμμών με τιμή μικρότερη ή ίση. Είναι χρήσιμα για ερωτήσεις του τύπου «σε ποιο εκατοστημόριο είναι αυτή η τιμή».

SQL

```
SELECT name, price,
       round(percent_rank() OVER (ORDER BY price)::numeric, 2) AS pct_rank,
       round(cume_dist() OVER (ORDER BY price)::numeric, 2) AS cume_dist
FROM products
WHERE category_id = 9
ORDER BY price;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name          | price  | pct_rank | cume_dist
-----------------------+--------+----------+-----------
 Ασύρματο ποντίκι      |  24.90 |     0.00 |      0.20
 Webcam HD             |  59.00 |     0.25 |      0.40
 Μηχανικό πληκτρολόγιο |  89.00 |     0.50 |      0.60
 USB-C docking station | 139.00 |     0.75 |      0.80
 Οθόνη 27" 4K          | 329.00 |     1.00 |      1.00
(5 rows)
```

ΚΡΑΤΑ ΑΥΤΟ

Με `ORDER BY` στο `OVER` κάθε γραμμή βλέπει ένα frame, από προεπιλογή από την αρχή μέχρι την ίδια και τις ισότιμές της. Το `ROWS` μετρά γραμμές και το `RANGE` τιμές. Το `LAG` και το `LEAD` διαβάζουν γειτονικές γραμμές. Το `last_value` θέλει frame που φτάνει ως το τέλος.

Ασκήσεις

ΑΣΚΗΣΗ 24.1 Σωρευτικά έσοδα ανά μήνα

Εμφάνισε για κάθε μήνα του 2026 τα έσοδα από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν και το σωρευτικό σύνολο από την αρχή του έτους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   month    | revenue  |   ytd
------------+----------+----------
 2026-01-01 |  1860.80 |  1860.80
 2026-02-01 |  3747.20 |  5608.00
 2026-03-01 |  1599.20 |  7207.20
 2026-04-01 |  4026.50 | 11233.70
 2026-05-01 | 10338.80 | 21572.50
 2026-06-01 |  9074.90 | 30647.40
(6 rows)
```

ΑΣΚΗΣΗ 24.2 Διαστήματα ανάμεσα σε αγορές

Για τον πελάτη 1 εμφάνισε κάθε παραγγελία με ημερομηνία και τις ημέρες από την προηγούμενη. Στην πρώτη παραγγελία δείξε 0 αντί για ∅.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  |    day     | days_since_previous
-----+------------+---------------------
  10 | 2025-03-18 |                   0
  11 | 2025-03-19 |                   1
  12 | 2025-03-25 |                   6
  17 | 2025-04-24 |                  30
  21 | 2025-05-08 |                  14
  29 | 2025-07-13 |                  66
  57 | 2025-11-19 |                 129
  58 | 2025-11-21 |                   2
  76 | 2026-01-10 |                  50
  91 | 2026-02-23 |                  44
 122 | 2026-05-18 |                  84
 129 | 2026-05-26 |                   8
(12 rows)
```

ΑΣΚΗΣΗ 24.3 Μεταβολή από μήνα σε μήνα

Ξαναγράψε τη μεταβολή των εσόδων από μήνα σε μήνα της θεωρίας με ένα CTE για τα μηνιαία έσοδα. Εμφάνισε μήνα, έσοδα, έσοδα του προηγούμενου μήνα και ποσοστό μεταβολής με ένα δεκαδικό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   month    | revenue  | previous | change_pct
------------+----------+----------+------------
 2026-01-01 |  1860.80 |        ∅ |          ∅
 2026-02-01 |  3747.20 |  1860.80 |      101.4
 2026-03-01 |  1599.20 |  3747.20 |      -57.3
 2026-04-01 |  4026.50 |  1599.20 |      151.8
 2026-05-01 | 10338.80 |  4026.50 |      156.8
 2026-06-01 |  9074.90 | 10338.80 |      -12.2
(6 rows)
```

ΑΣΚΗΣΗ 24.4 Εβδομαδιαίος κινούμενος μέσος

Για τις ημέρες του Ιουνίου 2026 με παραγγελίες, εμφάνισε τα έσοδα και τον μέσο όρο των εσόδων των ημερών με παραγγελίες μέσα στις επτά τελευταίες ημερολογιακές ημέρες, μαζί με την τρέχουσα. Κράτα τις γραμμές από τις 20 Ιουνίου και μετά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    day     | revenue | avg_7d
------------+---------+--------
 2026-06-21 |  688.80 | 373.48
 2026-06-23 |  751.40 | 428.68
 2026-06-24 | 1316.00 | 700.68
 2026-06-25 |  109.80 | 582.50
 2026-06-26 |  133.50 | 599.90
(5 rows)
```

ΑΣΚΗΣΗ 24.5 Πρώτη και τελευταία αγορά

Για τις παραγγελίες του πελάτη 6 εμφάνισε `id`, ημερομηνία, την ημερομηνία της πρώτης και της τελευταίας παραγγελίας του πελάτη, χωρίς `last_value`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id  |    day     | first_day  |  last_day
-----+------------+------------+------------
   2 | 2025-01-19 | 2025-01-19 | 2026-06-13
  25 | 2025-05-30 | 2025-01-19 | 2026-06-13
  26 | 2025-06-18 | 2025-01-19 | 2026-06-13
  39 | 2025-09-16 | 2025-01-19 | 2026-06-13
  42 | 2025-09-25 | 2025-01-19 | 2026-06-13
  45 | 2025-10-01 | 2025-01-19 | 2026-06-13
  55 | 2025-11-10 | 2025-01-19 | 2026-06-13
 102 | 2026-03-25 | 2025-01-19 | 2026-06-13
 121 | 2026-05-17 | 2025-01-19 | 2026-06-13
 143 | 2026-06-13 | 2025-01-19 | 2026-06-13
(10 rows)
```

ΑΣΚΗΣΗ 24.6 Νέοι και επαναλαμβανόμενοι πελάτες

Για κάθε μήνα του 2026 μέτρησε πόσες παραγγελίες ήταν η πρώτη του πελάτη τους και πόσες όχι. Λάβε υπόψη όλο το ιστορικό, όχι μόνο το 2026.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   month    | first_orders | repeat_orders
------------+--------------+---------------
 2026-01-01 |            2 |             4
 2026-02-01 |            1 |            11
 2026-03-01 |            2 |            10
 2026-04-01 |            1 |             8
 2026-05-01 |            0 |            20
 2026-06-01 |            0 |            29
(6 rows)
```

ΚΕΦΑΛΑΙΟ 25

Gaps and islands και χρονοσειρές

Μια χρονοσειρά από τον πίνακα έχει μόνο τις ημέρες όπου κάτι συνέβη. Για γραφήματα και κινούμενους μέσους όρους χρειάζεσαι και τις ημέρες με μηδέν. Για συνεχόμενες ημέρες δραστηριότητας και για συνεδρίες χρηστών χρειάζεσαι έναν τρόπο να ομαδοποιείς γραμμές που είναι συνεχόμενες. Και τα δύο λύνονται με εργαλεία που ήδη ξέρεις.

Συμπλήρωση κενών ημερών

Το `GROUP BY` ανά ημέρα δίνει μόνο ημέρες με παραγγελίες. Η λύση είναι να φτιάξεις πρώτα το ημερολόγιο με `generate_series`, όπως στο κεφάλαιο 6. Μετά ενώνεις τα δεδομένα σε αυτό με `LEFT JOIN`.

```sql
WITH days AS (
  SELECT d::date AS day
  FROM generate_series(date '2026-06-01', date '2026-06-14', interval '1 day') AS d
),
daily AS (
  SELECT o.created_at::date AS day, count(DISTINCT o.id) AS orders,
         sum(oi.quantity * oi.unit_price) AS revenue
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  WHERE o.status NOT IN ('cancelled', 'refunded')
  GROUP BY 1
)
SELECT days.day, COALESCE(daily.orders, 0) AS orders, COALESCE(daily.revenue, 0) AS revenue
FROM days
LEFT JOIN daily ON daily.day = days.day
ORDER BY days.day;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    day     | orders | revenue
------------+--------+---------
 2026-06-01 |      0 |       0
 2026-06-02 |      0 |       0
 2026-06-03 |      1 |   42.00
 2026-06-04 |      1 |  867.00
 2026-06-05 |      1 |  649.00
 2026-06-06 |      1 |  185.00
 2026-06-07 |      0 |       0
 2026-06-08 |      2 |   80.00
 2026-06-09 |      2 | 2353.50
 2026-06-10 |      0 |       0
 2026-06-11 |      0 |       0
 2026-06-12 |      0 |       0
 2026-06-13 |      2 |  367.90
 2026-06-14 |      4 |  725.90
(14 rows)
```

Το ημερολόγιο είναι ο αριστερός πίνακας, οπότε κάθε ημέρα εμφανίζεται. Το `COALESCE` βάζει μηδέν όπου δεν βρέθηκαν δεδομένα. Τώρα ένας κινούμενος μέσος όρος με `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW` μετρά πραγματικά επτά ημέρες, αφού κάθε ημέρα είναι μία γραμμή.

Το ίδιο κάνεις σε δύο διαστάσεις. Με `CROSS JOIN` ημερολογίου και κατηγοριών φτιάχνεις το πλήρες πλέγμα και με `LEFT JOIN` βάζεις τα δεδομένα μέσα.

SQL

```
WITH months AS (
  SELECT m::date AS month
  FROM generate_series(date '2026-04-01', date '2026-06-01', interval '1 month') AS m
),
sales AS (
  SELECT date_trunc('month', o.created_at)::date AS month, p.category_id,
         sum(oi.quantity) AS units
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  JOIN products AS p ON p.id = oi.product_id
  WHERE o.status NOT IN ('cancelled', 'refunded')
  GROUP BY 1, 2
)
SELECT cat.name AS category, m.month, COALESCE(s.units, 0) AS units
FROM months AS m
CROSS JOIN categories AS cat
LEFT JOIN sales AS s ON s.month = m.month AND s.category_id = cat.id
WHERE cat.id IN (8, 12)
ORDER BY cat.name, m.month;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 category |   month    | units
----------+------------+-------
 Κουζίνα  | 2026-04-01 |     0
 Κουζίνα  | 2026-05-01 |     2
 Κουζίνα  | 2026-06-01 |     7
 Laptops  | 2026-04-01 |     3
 Laptops  | 2026-05-01 |     5
 Laptops  | 2026-06-01 |     3
(6 rows)
```

Gaps

Ένα **gap** είναι ένα διάστημα χωρίς δραστηριότητα. Οι ημέρες του Ιουνίου χωρίς παραγγελίες βρίσκονται με το ημερολόγιο και ένα anti join. Τα διαστήματα ανάμεσα σε διαδοχικές ημέρες με παραγγελίες βρίσκονται πιο οικονομικά με `lead`, χωρίς ημερολόγιο.

SQL

```
WITH order_days AS (
  SELECT DISTINCT created_at::date AS day
  FROM orders
  WHERE created_at >= '2026-06-01'
)
SELECT day + 1 AS gap_start,
       next_day - 1 AS gap_end,
       next_day - day - 1 AS empty_days
FROM (SELECT day, lead(day) OVER (ORDER BY day) AS next_day FROM order_days) AS t
WHERE next_day - day > 1
ORDER BY gap_start;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 gap_start  |  gap_end   | empty_days
------------+------------+------------
 2026-06-07 | 2026-06-07 |          1
 2026-06-10 | 2026-06-12 |          3
 2026-06-16 | 2026-06-16 |          1
 2026-06-18 | 2026-06-18 |          1
 2026-06-20 | 2026-06-20 |          1
 2026-06-22 | 2026-06-22 |          1
 2026-06-27 | 2026-06-27 |          1
(7 rows)
```

Κάθε ημέρα με παραγγελίες κοιτάζει την επόμενη ημέρα με παραγγελίες. Αν απέχουν περισσότερο από μία ημέρα, ανάμεσά τους υπάρχει κενό.

Islands

Ένα **island** είναι ένα συνεχόμενο διάστημα δραστηριότητας. Ποιες είναι οι περίοδοι του Ιουνίου με παραγγελίες κάθε ημέρα; Το τέχνασμα είναι αριθμητικό. Αριθμείς τις ημέρες με παραγγελίες με `row_number` και αφαιρείς τον αριθμό από την ημερομηνία. Μέσα σε ένα island η ημερομηνία και ο αριθμός αυξάνονται μαζί κατά ένα, οπότε η διαφορά μένει σταθερή. Όταν υπάρχει κενό, η ημερομηνία πηδά ενώ ο αριθμός όχι. Τότε η διαφορά αλλάζει.

SQL

```
WITH order_days AS (
  SELECT DISTINCT created_at::date AS day
  FROM orders
  WHERE created_at >= '2026-06-01'
),
marked AS (
  SELECT day, day - row_number() OVER (ORDER BY day)::integer AS island
  FROM order_days
)
SELECT day, island
FROM marked
ORDER BY day
LIMIT 8;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    day     |   island
------------+------------
 2026-06-03 | 2026-06-02
 2026-06-04 | 2026-06-02
 2026-06-05 | 2026-06-02
 2026-06-06 | 2026-06-02
 2026-06-08 | 2026-06-03
 2026-06-09 | 2026-06-03
 2026-06-13 | 2026-06-06
 2026-06-14 | 2026-06-06
(8 rows)
```

Οι ημέρες 3 έως 6 Ιουνίου έχουν την ίδια τιμή στη στήλη `island`. Η τιμή δεν σημαίνει κάτι από μόνη της. Είναι απλώς μια ετικέτα κοινή για όλες τις ημέρες του ίδιου island. Ένα `GROUP BY` σε αυτήν δίνει τα islands.

SQL

```
WITH order_days AS (
  SELECT DISTINCT created_at::date AS day
  FROM orders
  WHERE created_at >= '2026-06-01'
),
marked AS (
  SELECT day, day - row_number() OVER (ORDER BY day)::integer AS island
  FROM order_days
)
SELECT min(day) AS island_start, max(day) AS island_end, count(*) AS days
FROM marked
GROUP BY island
ORDER BY island_start;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 island_start | island_end | days
--------------+------------+------
 2026-06-03   | 2026-06-06 |    4
 2026-06-08   | 2026-06-09 |    2
 2026-06-13   | 2026-06-15 |    3
 2026-06-17   | 2026-06-17 |    1
 2026-06-19   | 2026-06-19 |    1
 2026-06-21   | 2026-06-21 |    1
 2026-06-23   | 2026-06-26 |    4
 2026-06-28   | 2026-06-28 |    1
(8 rows)
```

Συνεδρίες

Μια παραλλαγή των islands είναι οι συνεδρίες χρηστών. Ο `events` καταγράφει κάθε κίνηση των πελατών στο site. Δύο διαδοχικά events ενός πελάτη ανήκουν στην ίδια συνεδρία αν απέχουν λιγότερο από 30 λεπτά. Εδώ δεν υπάρχει σταθερό βήμα για το τέχνασμα της αφαίρεσης, οπότε η τεχνική είναι άλλη. Πρώτα σημαδεύεις με 1 κάθε event που ξεκινά νέα συνεδρία. Μετά ένα τρέχον άθροισμα των σημαδιών δίνει τον αριθμό της συνεδρίας.

SQL

```
WITH flagged AS (
  SELECT customer_id, occurred_at, kind,
         CASE WHEN occurred_at - lag(occurred_at) OVER w <= interval '30 minutes'
              THEN 0 ELSE 1 END AS new_session
  FROM events
  WHERE customer_id = 13
  WINDOW w AS (PARTITION BY customer_id ORDER BY occurred_at)
),
sessions AS (
  SELECT *, sum(new_session) OVER (PARTITION BY customer_id ORDER BY occurred_at
                                   ROWS UNBOUNDED PRECEDING) AS session_no
  FROM flagged
)
SELECT session_no, min(occurred_at) AS started, count(*) AS events,
       string_agg(kind, ' → ' ORDER BY occurred_at) AS path
FROM sessions
GROUP BY session_no
ORDER BY session_no
LIMIT 6;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 session_no |        started         | events |                              path
------------+------------------------+--------
+-------------------------------------------------------------------
          1 | 2025-05-04 08:43:00+03 |      8 | search → view → cart → view → cart → view → cart → checkout
          2 | 2025-08-04 21:55:00+03 |      5 | search → view → view → search → view
          3 | 2025-09-03 08:35:00+03 |      8 | search → view → cart → view → cart → checkout → view → cart
          4 | 2025-09-19 20:00:00+03 |      9 | view → cart → view → cart → view → cart → view → cart → checkout
          5 | 2025-09-21 09:15:00+03 |      4 | search → view → cart → checkout
          6 | 2025-09-23 19:42:00+03 |      4 | search → search → view → search
(6 rows)
```

Το πρώτο event κάθε πελάτη δεν έχει προηγούμενο. Το `lag` δίνει `NULL`, η σύγκριση είναι άγνωστη και το `CASE` πέφτει στο `ELSE`, οπότε σωστά ξεκινά νέα συνεδρία. Το `ROWS UNBOUNDED PRECEDING` είναι συντομογραφία του `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

ΚΡΑΤΑ ΑΥΤΟ

Για χρονοσειρές χωρίς κενά φτιάχνεις το ημερολόγιο με `generate_series` και ενώνεις τα δεδομένα με `LEFT JOIN`. Τα gaps βρίσκονται με `lead`. Τα islands με σταθερό βήμα βρίσκονται με την αφαίρεση του `row_number`. Οι συνεδρίες βρίσκονται με σημάδι αρχής από `lag` και τρέχον άθροισμα.

Ασκήσεις

ΑΣΚΗΣΗ 25.1 Παραγγελίες κάθε ημέρας

Εμφάνισε για κάθε ημέρα από 20 έως 31 Μαΐου 2026 το πλήθος των παραγγελιών, οποιασδήποτε κατάστασης, με 0 για τις ημέρες χωρίς παραγγελίες.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    day     | orders
------------+--------
 2026-05-20 |      0
 2026-05-21 |      1
 2026-05-22 |      0
 2026-05-23 |      1
 2026-05-24 |      1
 2026-05-25 |      2
 2026-05-26 |      2
 2026-05-27 |      0
 2026-05-28 |      0
 2026-05-29 |      1
 2026-05-30 |      1
 2026-05-31 |      1
(12 rows)
```

ΑΣΚΗΣΗ 25.2 Εβδομάδες του 2026

Εμφάνισε τα έσοδα κάθε εβδομάδας του πρώτου τριμήνου του 2026, ξεκινώντας από τη Δευτέρα 29 Δεκεμβρίου 2025, από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν. Οι εβδομάδες χωρίς έσοδα πρέπει να έχουν 0.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    week    | revenue
------------+---------
 2025-12-29 |  324.90
 2026-01-05 |   79.90
 2026-01-12 |       0
 2026-01-19 | 1149.00
 2026-01-26 |  528.00
 2026-02-02 |  416.60
 2026-02-09 | 1584.00
 2026-02-16 |   45.60
 2026-02-23 | 1829.90
 2026-03-02 |  844.40
 2026-03-09 |       0
 2026-03-16 |  248.00
 2026-03-23 |  368.00
 2026-03-30 |   54.60
(14 rows)
```

ΑΣΚΗΣΗ 25.3 Κενά του Μαΐου

Βρες τα διαστήματα του Μαΐου 2026 χωρίς καμία παραγγελία, με αρχή, τέλος και πλήθος ημερών. Λάβε υπόψη μόνο παραγγελίες του Μαΐου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 gap_start  |  gap_end   | empty_days
------------+------------+------------
 2026-05-06 | 2026-05-06 |          1
 2026-05-08 | 2026-05-08 |          1
 2026-05-10 | 2026-05-10 |          1
 2026-05-13 | 2026-05-16 |          4
 2026-05-19 | 2026-05-20 |          2
 2026-05-22 | 2026-05-22 |          1
 2026-05-27 | 2026-05-28 |          2
(7 rows)
```

ΑΣΚΗΣΗ 25.4 Το μεγαλύτερο σερί

Βρες το μεγαλύτερο συνεχόμενο διάστημα ημερών του 2026 με τουλάχιστον μία παραγγελία. Εμφάνισε αρχή, τέλος και πλήθος ημερών. Αν υπάρχουν ισοβαθμίες, δείξε όλα τα διαστήματα με το μέγιστο μήκος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 island_start | island_end | days
--------------+------------+------
 2026-05-23   | 2026-05-26 |    4
 2026-06-03   | 2026-06-06 |    4
 2026-06-23   | 2026-06-26 |    4
(3 rows)
```

ΑΣΚΗΣΗ 25.5 Οι συνεδρίες του πελάτη 3

Για τον πελάτη 3 μέτρησε τις συνεδρίες με τον κανόνα των 30 λεπτών. Εμφάνισε το πλήθος των συνεδριών, τον μέσο αριθμό events ανά συνεδρία με ένα δεκαδικό και πόσες συνεδρίες περιέχουν `checkout`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 sessions | avg_events | with_checkout
----------+------------+---------------
       18 |        4.2 |             8
(1 row)
```

ΑΣΚΗΣΗ 25.6 Πελάτες που χάθηκαν για καιρό

Βρες τα διαστήματα πάνω από 120 ημέρες ανάμεσα σε δύο διαδοχικές παραγγελίες του ίδιου πελάτη. Εμφάνισε πελάτη, ημερομηνία της πρώτης, ημερομηνία της επόμενης και ημέρες, από το μεγαλύτερο διάστημα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id |  from_day  |   to_day   | days
-------------+------------+------------+------
           5 | 2025-02-07 | 2025-10-27 |  262
           3 | 2025-09-04 | 2026-04-23 |  231
          18 | 2025-11-05 | 2026-06-08 |  215
           4 | 2025-05-02 | 2025-11-26 |  208
           7 | 2025-10-17 | 2026-05-09 |  204
           9 | 2025-04-03 | 2025-10-19 |  199
           8 | 2025-03-28 | 2025-09-30 |  186
          17 | 2025-12-19 | 2026-06-08 |  171
          13 | 2025-09-21 | 2026-02-04 |  136
           6 | 2025-11-10 | 2026-03-25 |  135
           6 | 2025-01-19 | 2025-05-30 |  131
           1 | 2025-07-13 | 2025-11-19 |  129
          12 | 2025-07-18 | 2025-11-19 |  124
          14 | 2025-12-08 | 2026-04-10 |  123
…
(15 rows)
```

ΚΕΦΑΛΑΙΟ 26

GROUPING SETS, ROLLUP, CUBE και pivot

Μια αναφορά χρειάζεται συχνά λεπτομέρειες και σύνολα στο ίδιο αποτέλεσμα, ανά πόλη, ανά περιφέρεια και συνολικά. Το `GROUPING SETS` υπολογίζει πολλές ομαδοποιήσεις σε ένα query. Το pivot γυρίζει γραμμές σε στήλες. Τα ordered set aggregates δίνουν διάμεσο και επικρατούσα τιμή.

Πολλές ομαδοποιήσεις σε ένα query

Στο κεφάλαιο 12 έφτιαξες γραμμή συνόλου με `UNION ALL` και ένα δεύτερο query. Για σύνολα σε πολλά επίπεδα αυτό γίνεται γρήγορα δυσανάγνωστο. Το `GROUPING SETS` δέχεται μια λίστα από ομαδοποιήσεις και τις υπολογίζει όλες.

SQL

```
WITH sales AS (
  SELECT t.region, t.name AS city, oi.quantity * oi.unit_price AS amount
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  JOIN cities AS t ON t.id = o.shipping_city_id
  WHERE o.status NOT IN ('cancelled', 'refunded')
)
SELECT region, city, sum(amount) AS revenue
FROM sales
WHERE region IN ('Κρήτη', 'Θεσσαλία')
GROUP BY GROUPING SETS ((region, city), (region), ())
ORDER BY region, city;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  region  |   city   | revenue
----------+----------+----------
 Θεσσαλία | Βόλος    |  9190.60
 Θεσσαλία | Λάρισα   |  2410.10
 Θεσσαλία | ø        | 11600.70
 Κρήτη    | Ηράκλειο |  4089.80
 Κρήτη    | Χανιά    |  5521.80
 Κρήτη    | ø        |  9611.60
 ø        | ø        | 21212.30
(7 rows)
```

Κάθε παρένθεση της λίστας είναι μια ομαδοποίηση. Η `(region, city)` δίνει μία γραμμή ανά πόλη. Η `(region)` μία ανά περιφέρεια. Η κενή `()` μία γραμμή για όλα. Στις γραμμές όπου μια στήλη δεν

συμμετέχει στην ομαδοποίηση, η τιμή της είναι `NULL`. Γι' αυτό η γραμμή συνόλου έχει ∅ και στις δύο στήλες.

ROLLUP και CUBE

Η λίστα του προηγούμενου παραδείγματος είναι τόσο συνηθισμένη που έχει δικό της όνομα. Το `ROLLUP (a, b)` σημαίνει `GROUPING SETS ((a, b), (a), ())`, δηλαδή τα επίπεδα μιας ιεραρχίας από το λεπτομερές στο συνολικό.

Το `CUBE (a, b)` δίνει όλους τους συνδυασμούς, `(a, b)`, `(a)`, `(b)` και `()`. Είναι χρήσιμο για διαστάσεις που δεν είναι ιεραρχία, όπως η μέθοδος πληρωμής και το έτος.

SQL

```
SELECT method, extract(year FROM paid_at) AS year, sum(amount) AS total
FROM payments
WHERE method IN ('card', 'paypal')
GROUP BY CUBE (method, extract(year FROM paid_at))
ORDER BY method, year;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 method | year |  total
--------+------+----------
 card   | 2025 | 20673.50
 card   | 2026 | 16232.00
 card   |    ∅ | 36905.50
 paypal | 2025 |  4798.50
 paypal | 2026 |  5178.90
 paypal |    ∅ |  9977.40
 ∅      | 2025 | 25472.00
 ∅      | 2026 | 21410.90
 ∅      |    ∅ | 46882.90
(9 rows)
```

Οι γραμμές με ∅ στο έτος είναι τα σύνολα της μεθόδου. Οι γραμμές με ∅ στη μέθοδο είναι τα σύνολα του έτους. Η γραμμή με ∅ και στα δύο είναι το γενικό σύνολο.

GROUPING

Ένα πρόβλημα μένει. Αν τα δεδομένα έχουν `NULL`, για παράδειγμα πόλη αποστολής που λείπει, δεν ξέρεις αν το ∅ μιας γραμμής είναι η τιμή ή σημάδι συνόλου. Η συνάρτηση `GROUPING(στήλη)` δίνει 1 όταν η στήλη δεν συμμετέχει στην ομαδοποίηση της γραμμής και 0 όταν συμμετέχει. Με αυτήν γράφεις ετικέτες.

SQL

```
SELECT CASE WHEN GROUPING(method) = 1 THEN 'όλες' ELSE method END AS method,
       CASE WHEN GROUPING(extract(year FROM paid_at)) = 1 THEN 'όλα'
            ELSE extract(year FROM paid_at)::text END AS year,
       sum(amount) AS total
FROM payments
WHERE method IN ('card', 'paypal')
GROUP BY ROLLUP (method, extract(year FROM paid_at))
ORDER BY GROUPING(method), method, GROUPING(extract(year FROM paid_at)), 2;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 method | year |  total
--------+------+----------
 card   | 2025 | 20673.50
 card   | 2026 | 16232.00
 card   | όλα  | 36905.50
 paypal | 2025 |  4798.50
 paypal | 2026 |  5178.90
 paypal | όλα  |  9977.40
 όλες   | όλα  | 46882.90
(7 rows)
```

Το `GROUPING` μπαίνει και στο `ORDER BY`, ώστε τα σύνολα να έρχονται μετά τις λεπτομέρειες. Το `2` ταξινομεί με τη δεύτερη στήλη του αποτελέσματος, εδώ το έτος ως κείμενο.

Pivot με FILTER

Το **pivot** γυρίζει τις τιμές μιας στήλης σε στήλες. Αντί για μία γραμμή ανά μήνα και κατάσταση θέλεις μία γραμμή ανά μήνα και μία στήλη ανά κατάσταση. Το `FILTER` του κεφαλαίου 11 το κάνει άμεσα.

SQL

```
SELECT to_char(created_at, 'YYYY-MM') AS month,
       count(*) FILTER (WHERE status = 'delivered') AS delivered,
       count(*) FILTER (WHERE status = 'shipped') AS shipped,
       count(*) FILTER (WHERE status IN ('pending', 'paid')) AS open,
       count(*) FILTER (WHERE status IN ('cancelled', 'refunded')) AS lost,
       count(*) AS total
FROM orders
WHERE created_at >= '2026-01-01'
GROUP BY 1
ORDER BY 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  month  | delivered | shipped | open | lost | total
---------+-----------+---------+------+------+-------
 2026-01 |         6 |       0 |    0 |    0 |     6
 2026-02 |        11 |       0 |    0 |    1 |    12
 2026-03 |        10 |       0 |    0 |    2 |    12
 2026-04 |         8 |       0 |    0 |    1 |     9
 2026-05 |        18 |       0 |    0 |    2 |    20
 2026-06 |         9 |      12 |    8 |    0 |    29
(6 rows)
```

Οι στήλες του pivot γράφονται στο query. Αν αύριο προστεθεί νέα κατάσταση, πρέπει να προσθέσεις στήλη. Η SQL δεν μπορεί να φτιάξει στήλες δυναμικά από τα δεδομένα, αφού ο αριθμός και οι τύποι των στηλών πρέπει να είναι γνωστοί πριν από την εκτέλεση. Η επέκταση `tablefunc` έχει τη συνάρτηση `crosstab`, αλλά κι αυτή απαιτεί να δηλώσεις τις στήλες.

Διάμεσος και επικρατούσα τιμή

Ο μέσος όρος παραπλανά όταν υπάρχουν λίγες πολύ μεγάλες τιμές. Μερικά laptops ανεβάζουν τη μέση αξία παραγγελίας πολύ πάνω από μια τυπική παραγγελία. Η **διάμεσος** είναι η τιμή στη μέση της ταξινομημένης λίστας. Στην PostgreSQL υπολογίζεται με `percentile_cont(0.5) WITHIN GROUP (ORDER BY ...)`.

```sql
WITH totals AS (
  SELECT o.status, sum(oi.quantity * oi.unit_price) AS total
  FROM orders AS o
  JOIN order_items AS oi ON oi.order_id = o.id
  GROUP BY o.id
)
SELECT status, count(*) AS orders,
       round(avg(total), 2) AS mean,
       round(percentile_cont(0.5) WITHIN GROUP (ORDER BY total)::numeric, 2) AS median,
       round(percentile_cont(0.9) WITHIN GROUP (ORDER BY total)::numeric, 2) AS p90
FROM totals
GROUP BY status
ORDER BY orders DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  status   | orders |  mean  | median |   p90
-----------+--------+--------+--------+---------
 delivered |    125 | 478.62 | 212.00 | 1207.22
 cancelled |     14 | 252.00 | 115.25 |  719.50
 shipped   |     12 | 167.06 | 154.60 |  234.30
 paid      |      5 | 281.06 | 133.50 |  589.28
 refunded  |      3 | 539.67 | 339.00 | 1019.80
 pending   |      2 | 697.60 | 697.60 | 1192.32
(6 rows)
```

Στις παραδομένες παραγγελίες η διάμεσος είναι σχεδόν το μισό του μέσου όρου. Η τυπική παραγγελία είναι πολύ μικρότερη από όσο δείχνει ο μέσος όρος. Το `percentile_cont` επιστρέφει `double precision`, γι' αυτό η μετατροπή σε `numeric` πριν από το `round`.

Αυτές λέγονται **ordered set aggregates**, γιατί χρειάζονται τις τιμές ταξινομημένες, με τη σειρά που ορίζει το `WITHIN GROUP`. Το `percentile_disc` επιστρέφει πάντα μια πραγματική τιμή από τα δεδομένα, ενώ το `percentile_cont` παρεμβάλλει ανάμεσα σε δύο τιμές. Το `mode()` δίνει την πιο συχνή τιμή.

```sql
SELECT t.region, mode() WITHIN GROUP (ORDER BY pay.method) AS usual_method
FROM payments AS pay
JOIN orders AS o ON o.id = pay.order_id
JOIN cities AS t ON t.id = o.shipping_city_id
WHERE pay.amount > 0
GROUP BY t.region
ORDER BY t.region;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       region       | usual_method
--------------------+--------------
 Αττική             | card
 Δυτική Ελλάδα      | card
 Ήπειρος            | paypal
 Θεσσαλία           | card
 Κεντρική Μακεδονία | card
 Κρήτη              | card
 Νότιο Αιγαίο       | card
 Πελοπόννησος       | card
(8 rows)
```

ΚΡΑΤΑ ΑΥΤΟ

Το `GROUPING SETS` υπολογίζει πολλές ομαδοποιήσεις σε ένα πέρασμα. Το `ROLLUP` δίνει επίπεδα ιεραρχίας και το `CUBE` όλους τους συνδυασμούς. Το `GROUPING()` ξεχωρίζει τις γραμμές συνόλων. Το pivot γράφεται με ένα `FILTER` ανά στήλη. Η διάμεσος είναι `percentile_cont(0.5) WITHIN GROUP`.

Ασκήσεις

ΑΣΚΗΣΗ 26.1 Υπάλληλοι ανά τμήμα και θέση

Εμφάνισε το πλήθος και το άθροισμα μισθών των υπαλλήλων με τμήμα, ανά τμήμα και θέση, με υποσύνολα ανά τμήμα και γενικό σύνολο. Χρησιμοποίησε `ROLLUP`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 department |       title        | employees | payroll
------------+--------------------+-----------+----------
 Αποθήκη    | Αποθηκάριος        |         2 |  3550.00
 Αποθήκη    | Οδηγός διανομής    |         1 |  1650.00
 Αποθήκη    | Operations Manager |         1 |  6200.00
 Αποθήκη    | ∅                  |         4 | 11400.00
 Διοίκηση   | CEO                |         1 |  9500.00
 Διοίκηση   | ∅                  |         1 |  9500.00
 Πωλήσεις   | Account Manager    |         2 |  6000.00
 Πωλήσεις   | Head of Sales      |         1 |  7000.00
 Πωλήσεις   | Sales Intern       |         1 |  1100.00
 Πωλήσεις   | ∅                  |         4 | 14100.00
 Τεχνολογία | Backend Developer  |         1 |  4200.00
 Τεχνολογία | CTO                |         1 |  8200.00
 Τεχνολογία | Data Engineer      |         1 |  4400.00
 Τεχνολογία | Frontend Developer |         1 |  3900.00
 …
(17 rows)
```

ΑΣΚΗΣΗ 26.2 Ετικέτες συνόλων

Ξαναγράψε την προηγούμενη άσκηση ώστε οι γραμμές συνόλων να γράφουν `όλα τα τμήματα` και `όλες οι θέσεις` αντί για ∅. Κράτα μόνο τις στήλες τμήματος, θέσης και πλήθους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 department |       title        | employees
------------+--------------------+-----------
 Αποθήκη    | Αποθηκάριος        |         2
 Αποθήκη    | Οδηγός διανομής    |         1
 Αποθήκη    | Operations Manager |         1
 Αποθήκη    | όλες οι θέσεις     |         4
 Διοίκηση   | CEO                |         1
 Διοίκηση   | όλες οι θέσεις     |         1
 Πωλήσεις   | Account Manager    |         2
 Πωλήσεις   | Head of Sales      |         1
 Πωλήσεις   | Sales Intern       |         1
 Πωλήσεις   | όλες οι θέσεις     |         4
 Τεχνολογία | Backend Developer  |         1
 Τεχνολογία | CTO                |         1
 Τεχνολογία | Data Engineer      |         1
 Τεχνολογία | Frontend Developer |         1
 …
(17 rows)
```

ΑΣΚΗΣΗ 26.3 Κύβος πληρωμών

Με `CUBE`, υπολόγισε το πλήθος των θετικών πληρωμών ανά μέθοδο και ανά τρίμηνο του 2026, με όλους τους συνδυασμούς συνόλων. Το τρίμηνο να είναι αριθμός από 1 έως 4.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  method  | quarter | payments
----------+---------+----------
 bank     |       1 |        1
 bank     |       2 |        6
 bank     |       ø |        7
 card     |       1 |       19
 card     |       2 |       31
 card     |       ø |       50
 cod      |       1 |        4
 cod      |       2 |        7
 cod      |       ø |       11
 giftcard |       1 |        3
 giftcard |       2 |        3
 giftcard |       ø |        6
 paypal   |       1 |        4
 paypal   |       2 |        8
 …
(18 rows)
```

ΑΣΚΗΣΗ 26.4 Πίνακας μηνών και μεθόδων

Φτιάξε pivot με μία γραμμή ανά μήνα του 2026 και μία στήλη για το καθαρό ποσό κάθε μεθόδου πληρωμής `card`, `paypal`, `bank` και `cod`. Όπου δεν υπάρχει ποσό γράψε 0.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  month  |  card   | paypal  |  bank   |   cod
---------+---------+---------+---------+---------
 2026-01 |  278.90 | 1149.00 |       0 |  103.90
 2026-02 | 2357.70 |  128.00 |       0 | 1540.50
 2026-03 | 1366.30 |   30.40 |   20.50 |   82.00
 2026-04 | 1043.70 | 2208.90 |  773.90 |       0
 2026-05 | 6722.70 |  649.00 | 2400.10 |  567.00
 2026-06 | 4462.70 | 1013.60 |  817.00 | 1081.70
(6 rows)
```

ΑΣΚΗΣΗ 26.5 Διάμεσος ανά κατηγορία

Για κάθε κατηγορία με ενεργά προϊόντα εμφάνισε τη μέση και τη διάμεσο τιμή τους, με δύο δεκαδικά. Εμφάνισε μόνο τις κατηγορίες όπου ο μέσος όρος ξεπερνά τη διάμεσο κατά περισσότερο από 10%.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   category   |  mean   | median
--------------+---------+---------
 Κουζίνα      |  115.97 |   64.00
 Περιφερειακά |  128.18 |   89.00
 Laptops      | 1332.33 | 1149.00
(3 rows)
```

ΑΣΚΗΣΗ 26.6 Η συνηθισμένη ώρα

Για κάθε ημέρα της εβδομάδας, με 1 τη Δευτέρα, βρες την ώρα της ημέρας με τις περισσότερες παραγγελίες, με `mode()`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 weekday | busiest_hour | orders
---------+--------------+--------
       1 |           14 |     31
       2 |            9 |     23
       3 |           11 |     21
       4 |            8 |     22
       5 |           13 |     22
       6 |           22 |     15
       7 |           22 |     28
(7 rows)
```

ΚΕΦΑΛΑΙΟ 27

Arrays και JSONB

Η PostgreSQL μπορεί να κρατήσει μια λίστα τιμών ή ένα ολόκληρο έγγραφο JSON σε μία στήλη. Είναι χρήσιμο για ετικέτες, για χαρακτηριστικά που διαφέρουν από προϊόν σε προϊόν και για δεδομένα που έρχονται ήδη ως JSON. Δεν αντικαθιστά τους πίνακες, αλλά τους συμπληρώνει.

Arrays

Η στήλη `tags` του `products` είναι `text[]`, ένας πίνακας από κείμενα. Ένα array γράφεται ως κείμενο με άγκιστρα, `'{a,b}'`, ή με τον constructor `ARRAY['a', 'b']`. Τα στοιχεία αριθμούνται από το 1.

SQL

```
SELECT name, tags, tags[1] AS first_tag, cardinality(tags) AS n_tags
FROM products
WHERE id IN (1, 19, 28);
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                |              tags               | first_tag | n_tags
------------------------------------+---------------------------------+-----------+--------
 Βίος και πολιτεία του Αλέξη Ζορμπά | {βιβλίο,λογοτεχνία,bestseller} | βιβλίο    |      3
 Ακουστικά ANC                      | {ήχος,ασύρματο,bestseller}     | ήχος      |      3
 Δωροκάρτα 50€                      | {δώρο}                         | δώρο      |      1
(3 rows)
```

Για να ρωτήσεις αν ένα array περιέχει μια τιμή υπάρχουν δύο τρόποι. Το `'τιμή' = ANY(tags)` είναι αληθές όταν κάποιο στοιχείο είναι ίσο με την τιμή. Ο τελεστής `@>` ελέγχει αν το αριστερό array περιέχει όλα τα στοιχεία του δεξιού. Το `&&` ελέγχει αν έχουν τουλάχιστον ένα κοινό.

SQL

```
SELECT name, tags
FROM products
WHERE tags @> ARRAY['ασύρματο']
   OR tags && ARRAY['podcast', 'βινύλιο']
ORDER BY name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name       |            tags
------------------+----------------------------
 Ακουστικά ANC    | {ήχος,ασύρματο,bestseller}
 Ασύρματο ποντίκι | {περιφερειακά,ασύρματο}
 Ηχείο Bluetooth  | {ήχος,ασύρματο}
 Μικρόφωνο USB    | {ήχος,podcast}
 Πικάπ            | {ήχος,βινύλιο}
(5 rows)
```

Ο `@>` και ο `&&` μπορούν να χρησιμοποιήσουν GIN index, κάτι που θα δούμε στο κεφάλαιο 35. Το `= ANY` δεν μπορεί.

unnest και array_agg

Το `unnest` κάνει ένα array γραμμές. Είναι το κλειδί για κάθε ανάλυση πάνω σε arrays, γιατί μετά δουλεύεις με κανονικές γραμμές και `GROUP BY`. Στο `FROM` μπαίνει με `CROSS JOIN LATERAL`, όπως στο κεφάλαιο 14, αφού παίρνει όρισμα από την προηγούμενη γραμμή. Η PostgreSQL δέχεται και την πιο σύντομη γραφή με κόμμα, που για συναρτήσεις είναι πάντα lateral.

SQL

```
SELECT tag, count(*) AS products
FROM products AS p
CROSS JOIN LATERAL unnest(p.tags) AS tag
WHERE p.active
GROUP BY tag
HAVING count(*) >= 3
ORDER BY products DESC, tag;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       tag       | products
-----------------+----------
 βιβλίο          |       10
 περιφερειακά    |        5
 ήχος            |        4
 λογοτεχνία      |        4
 ασύρματο        |        3
 γραφείο         |        3
 προγραμματισμός |        3
 σπίτι           |        3
 laptop          |        3
(9 rows)
```

Το αντίστροφο είναι το `array_agg`, που μαζεύει τιμές μιας ομάδας σε array. Με `WITH ORDINALITY` το `unnest` δίνει και τη θέση κάθε στοιχείου.

SQL

```
SELECT p.name, t.tag, t.pos
FROM products AS p
CROSS JOIN LATERAL unnest(p.tags) WITH ORDINALITY AS t(tag, pos)
WHERE p.id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                name                |    tag     | pos
------------------------------------+------------+-----
 Βίος και πολιτεία του Αλέξη Ζορμπά | βιβλίο     |   1
 Βίος και πολιτεία του Αλέξη Ζορμπά | λογοτεχνία |   2
 Βίος και πολιτεία του Αλέξη Ζορμπά | bestseller |   3
(3 rows)
```

ΠΑΓΙΔΑ

Ένα array είναι βολικό για λίστες που διαβάζονται ολόκληρες, όπως οι ετικέτες ενός προϊόντος. Δεν μπορεί όμως να έχει foreign key στα στοιχεία του ούτε constraint μοναδικότητας ανάμεσα σε γραμμές. Αν τα στοιχεία είναι οντότητες με δικά τους στοιχεία, όπως προμηθευτές, χρειάζεσαι πίνακα σύνδεσης όπως στο κεφάλαιο 18.

jsonb

Η στήλη `attrs` είναι `jsonb`. Κρατά ένα έγγραφο JSON σε δυαδική μορφή, με κλειδιά που διαφέρουν από προϊόν σε προϊόν. Ένα βιβλίο έχει σελίδες και εκδότη, ένα ποντίκι ανάλυση και χρώματα.

SQL

```
SELECT name, attrs
FROM products
WHERE id IN (5, 14);
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name       |                            attrs
------------------+-------------------------------------------------------------
 Μαθαίνω Python   | {"pages": 520, "language": "el", "publisher": "Κλειδάριθμος"}
 Ασύρματο ποντίκι | {"dpi": 1600, "colors": ["black", "white"], "wireless": true}
(2 rows)
```

Η PostgreSQL έχει και τον τύπο `json`, που κρατά το κείμενο όπως είναι. Ο `jsonb` αφαιρεί κενά και διπλά κλειδιά, ταξινομεί τα κλειδιά και διαβάζεται πολύ γρηγορότερα. Χρησιμοποίησε `jsonb`, εκτός αν πρέπει να διατηρηθεί το ακριβές κείμενο.

Διαβάζοντας τιμές

Ο τελεστής `->` επιστρέφει την τιμή ενός κλειδιού ως `jsonb`. Ο `->>` την επιστρέφει ως `text`. Ο `#>>` ακολουθεί ένα μονοπάτι. Για σύγκριση ή πράξεις με αριθμούς μετατρέπεις το κείμενο στον τύπο που θέλεις.

SQL

```
SELECT name,
       attrs -> 'colors' AS colors_json,
       attrs ->> 'ram_gb' AS ram_text,
       (attrs ->> 'ram_gb')::integer * 1024 AS ram_mb,
       attrs #>> '{colors,0}' AS first_color
FROM products
WHERE category_id = 8
ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name        |     colors_json     | ram_text | ram_mb | first_color
-------------------+---------------------+----------+--------+-------------
 Laptop Aero 14    | ["silver", "black"] | 16       |  16384 | silver
 Laptop Pro 16     | ["space gray"]      | 32       |  32768 | space gray
 Laptop Student 15 | ["black"]           | 8        |   8192 | black
 Laptop Classic 13 | ∅                   | 8        |   8192 | ∅
(4 rows)
```

Στα arrays του JSON η αρίθμηση ξεκινά από το 0, σε αντίθεση με τα arrays της PostgreSQL. Όπου το κλειδί λείπει, το αποτέλεσμα είναι `NULL`, όπως στο `Laptop Classic 13` που δεν έχει χρώματα.

Για φιλτράρισμα υπάρχουν τελεστές που ρωτούν τη δομή. Το `?` ελέγχει αν υπάρχει κλειδί. Το `@>` ελέγχει αν το έγγραφο περιέχει ένα μικρότερο έγγραφο.

SQL

```
SELECT name, attrs ->> 'battery_h' AS battery
FROM products
WHERE attrs ? 'battery_h'
  AND attrs @> '{"wireless": true}'
ORDER BY name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
      name       | battery
-----------------+---------
 Ακουστικά ANC   | 30
 Ηχείο Bluetooth | 12
(2 rows)
```

Το `@>` είναι ο τελεστής που θέλεις σε μεγάλους πίνακες. Όπως στα arrays, χρησιμοποιεί GIN index. Ένα `WHERE (attrs ->> 'wireless')::boolean` δίνει το ίδιο αποτέλεσμα αλλά χωρίς index.

JSON σε γραμμές

Το `jsonb_array_elements_text` κάνει ένα JSON array γραμμές, όπως το `unnest`. Το `jsonb_each_text` κάνει ένα αντικείμενο γραμμές κλειδιού και τιμής.

SQL

```
SELECT p.name, kv.key, kv.value
FROM products AS p
CROSS JOIN LATERAL jsonb_each_text(p.attrs) AS kv
WHERE p.id = 16;
```

ΑΠΟΤΕΛΕΣΜΑ

```
     name     |    key     |       value
--------------+------------+-------------------
 Οθόνη 27" 4K | ports      | ["hdmi", "usb-c"]
 Οθόνη 27" 4K | screen_in  | 27
 Οθόνη 27" 4K | resolution | 3840x2160
(3 rows)
```

Από την PostgreSQL 17 υπάρχει και το `JSON_TABLE` του προτύπου, που μετατρέπει JSON σε γραμμές και στήλες με τύπους σε ένα βήμα. Τα μονοπάτια γράφονται σε **jsonpath**, μια μικρή γλώσσα όπου το `$` είναι η ρίζα του εγγράφου.

SQL

```
SELECT p.name, c.color
FROM products AS p
CROSS JOIN LATERAL JSON_TABLE(p.attrs, '$.colors[*]'
                              COLUMNS (color text PATH '$')) AS c
WHERE p.category_id IN (8, 9)
ORDER BY p.name, c.color;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name        |   color
-------------------+------------
 Ασύρματο ποντίκι  | black
 Ασύρματο ποντίκι  | white
 Laptop Aero 14    | black
 Laptop Aero 14    | silver
 Laptop Pro 16     | space gray
 Laptop Student 15 | black
(6 rows)
```

Φτιάχνοντας JSON

Συχνά η εφαρμογή θέλει το αποτέλεσμα ως JSON. Το `jsonb_build_object` φτιάχνει αντικείμενο από ζεύγη κλειδιών και τιμών και το `jsonb_agg` μαζεύει τις τιμές μιας ομάδας σε array. Μαζί φτιάχνουν ένθετα έγγραφα σε ένα query.

SQL

```
SELECT jsonb_pretty(jsonb_build_object(
         'customer', c.first_name || ' ' || c.last_name,
         'orders', jsonb_agg(jsonb_build_object('id', o.id, 'status', o.status)
                             ORDER BY o.id))) AS doc
FROM customers AS c
JOIN orders AS o ON o.customer_id = c.id
WHERE c.id = 17
GROUP BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                doc
-----------------------------------
 {                                 +
     "orders": [                   +
         {                         +
             "id": 47,             +
             "status": "delivered" +
         },                        +
         {                         +
             "id": 68,             +
             "status": "cancelled" +
         },                        +
         {                         +
```

```
            "id": 138,            +
            "status": "delivered" +
        },                        +
        {                         +
            "id": 154,            +
            "status": "shipped"   +
        }                         +
    ],                            +
    "customer": "Φωτεινή Λαμπράκη"+
}
(1 row)
```

Το `jsonb_pretty` είναι μόνο για εμφάνιση. Βάζει αλλαγές γραμμής και εσοχές.

Αλλάζοντας JSON

Ο τελεστής `||` ενώνει δύο αντικείμενα. Όπου υπάρχει ίδιο κλειδί, κερδίζει το δεξί. Ο τελεστής `-` αφαιρεί κλειδί. Το `jsonb_set` αλλάζει μια τιμή σε μονοπάτι.

SQL

```
UPDATE products
SET attrs = attrs || '{"warranty_years": 2}'
WHERE category_id = 8
RETURNING name, attrs ->> 'warranty_years' AS warranty;
```

ΑΠΟΤΕΛΕΣΜΑ

```
       name        | warranty
-------------------+----------
 Laptop Aero 14    | 2
 Laptop Pro 16     | 2
 Laptop Student 15 | 2
 Laptop Classic 13 | 2
(4 rows)

UPDATE 4
```

Ένα `UPDATE` σε jsonb ξαναγράφει ολόκληρο το έγγραφο, όχι μόνο το κλειδί που άλλαξε. Για μεγάλα έγγραφα που αλλάζουν συχνά, αυτό κοστίζει.

ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL

Αν ένα κλειδί του JSON εμφανίζεται σε όλες τις γραμμές, χρησιμοποιείται σε φίλτρα και joins ή πρέπει να έχει constraint, ανήκει σε δική του στήλη. Το jsonb είναι για δεδομένα που πραγματικά διαφέρουν από γραμμή σε γραμμή ή που δεν σου ανήκουν, όπως το σώμα ενός webhook.

ΚΡΑΤΑ ΑΥΤΟ

Τα arrays κρατούν λίστες ίδιου τύπου, αριθμημένες από το 1. Το `unnest` τα κάνει γραμμές και το `array_agg` το αντίστροφο. Στο jsonb το `->` δίνει jsonb, το `->>` κείμενο και το `@>` ελέγχει περιεχόμενο με υποστήριξη index. Το `JSON_TABLE` και το `jsonb_agg` μετατρέπουν ανάμεσα σε JSON και γραμμές.

Ασκήσεις

ΑΣΚΗΣΗ 27.1 Προϊόντα γραφείου

Βρες τα προϊόντα που έχουν την ετικέτα `γραφείο` ή την ετικέτα `usb-c`. Εμφάνισε όνομα και ετικέτες.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             |         tags
-----------------------------+----------------------
 Γραφείο ρυθμιζόμενου ύψους | {γραφείο,εργονομία}
 Εργονομική καρέκλα         | {γραφείο,εργονομία}
 Φωτιστικό γραφείου LED     | {γραφείο}
 USB-C docking station      | {περιφερειακά,usb-c}
(4 rows)
```

ΑΣΚΗΣΗ 27.2 Πωλήσεις ανά ετικέτα

Υπολόγισε τα τεμάχια που πουλήθηκαν για κάθε ετικέτα προϊόντος σε παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν. Εμφάνισε τις πέντε ετικέτες με τα περισσότερα τεμάχια.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
       tag       | units
-----------------+-------
 βιβλίο          |   135
 λογοτεχνία      |    56
 περιφερειακά    |    48
 προγραμματισμός |    39
 γραφείο         |    36
(5 rows)
```

ΑΣΚΗΣΗ 27.3 Laptops με μνήμη

Εμφάνισε όνομα, μνήμη και αποθηκευτικό χώρο των laptops με τουλάχιστον 16 GB μνήμη, ως ακέραιους, ταξινομημένα από τη μεγαλύτερη μνήμη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      name      | ram_gb | storage_gb
----------------+--------+------------
 Laptop Pro 16  |     32 |       1024
 Laptop Aero 14 |     16 |        512
(2 rows)
```

ΑΣΚΗΣΗ 27.4 Οι συχνότερες αναζητήσεις

Από τα events τύπου `search`, βρες τις πέντε συχνότερες αναζητήσεις με το πλήθος τους και πόσοι διαφορετικοί πελάτες τις έκαναν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      query       | searches | customers
------------------+----------+-----------
 postgres         |       26 |        18
 ακουστικά ANC    |       24 |        14
 καρεκλα γραφειου |       23 |        15
 καφετιερα        |       22 |        12
 python           |       20 |        19
(5 rows)
```

ΑΣΚΗΣΗ 27.5 Πελάτης ως JSON

Φτιάξε για τον πελάτη 13 ένα έγγραφο JSON με το ονοματεπώνυμο, την πόλη και ένα array με τις παραγγελίες του, όπου κάθε παραγγελία έχει `id` και ημερομηνία ως κείμενο `YYYY-MM-DD`. Εμφάνισέ το με `jsonb_pretty`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
               doc
----------------------------------
 {                                +
     "city": "Χανιά",             +
     "name": "Κατερίνα Σταύρου", +
     "orders": [                  +
         {                        +
             "id": 20,            +
             "date": "2025-05-04"+
         },                       +
         {                        +
             "id": 35,            +
             "date": "2025-09-03"+
         },                       +
         {                        +
             "id": 40,            +
             "date": "2025-09-19"+
         },                       +
         {                        +
             "id": 41,            +
             "date": "2025-09-21"+
         },                       +
         {                        +
             "id": 82,            +
             "date": "2026-02-04"+
         }                        +
     ]                            +
 }
(1 row)
```

ΑΣΚΗΣΗ 27.6 Θύρες σε γραμμές

Με `JSON_TABLE`, εμφάνισε για κάθε προϊόν με κλειδί `ports` το όνομά του και κάθε θύρα σε δική της γραμμή, με τη θέση της στη λίστα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
         name           | pos |   port
------------------------+-----+----------
 Οθόνη 27" 4K           |   1 | hdmi
 Οθόνη 27" 4K           |   2 | usb-c
 USB-C docking station  |   1 | hdmi
 USB-C docking station  |   2 | ethernet
 USB-C docking station  |   3 | usb-a
 USB-C docking station  |   4 | usb-c
(6 rows)
```

ΑΣΚΗΣΗ 27.7 Εγγύηση και αφαίρεση κλειδιού

Σε ένα query, πρόσθεσε στα προϊόντα της κατηγορίας 10 το κλειδί `warranty_years` με τιμή 1 και αφαίρεσε από όλα τα προϊόντα το κλειδί `colors`. Εμφάνισε για τις γραμμές της κατηγορίας 10 που άλλαξαν το όνομα, την εγγύηση και αν έχουν ακόμη κλειδί `colors`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      name       | warranty | has_colors
-----------------+----------+------------
 Ακουστικά ANC   | 1        | f
 Ηχείο Bluetooth | 1        | f
 Μικρόφωνο USB   | 1        | f
 Πικάπ           | 1        | f
(4 rows)
```

ΚΕΦΑΛΑΙΟ 28

Αναζήτηση κειμένου με regex, trigrams και full text search

Το `LIKE` βρίσκει μοτίβα, αλλά δεν αντέχει τόνους, ορθογραφικά λάθη ή διαφορετικούς τύπους της ίδιας λέξης. Η PostgreSQL έχει τρία εργαλεία για αυτά. Τα regular expressions για ακριβή μοτίβα, τα trigrams για ομοιότητα και το full text search για λέξεις με κατάταξη αποτελεσμάτων.

Regular expressions

Ο τελεστής `~` ελέγχει αν ένα κείμενο ταιριάζει με regular expression. Ο `~*` κάνει το ίδιο χωρίς διάκριση πεζών και κεφαλαίων. Ο `!~` είναι η άρνηση. Τα regular expressions περιγράφουν μοτίβα που το `LIKE` δεν μπορεί, όπως «ακριβώς δέκα ψηφία που ξεκινούν με 69».

SQL

```
SELECT name, phone
FROM suppliers
WHERE phone !~ '^69[0-9]{8}$';
```

ΑΠΟΤΕΛΕΣΜΑ

```
         name           |    phone
------------------------+--------------
 Βιβλιοδιανομή Αττικής | 2105550101
 Techno Import         | 2310555202
 Ήχος & Εικόνα ΑΕ      | 2105550303
 Nordic Office         | +46855500404
 Θεσσαλική Χαρτική     | 2410555606
(5 rows)
```

Το `^` σημαίνει αρχή, το `$` τέλος, το `[0-9]` ένα ψηφίο και το `{8}` ακριβώς οκτώ φορές. Κανένας προμηθευτής δεν έχει κινητό. Η γραμμή με `NULL` τηλέφωνο δεν εμφανίζεται, για τον γνωστό λόγο του κεφαλαίου 3.

Οι συναρτήσεις regex εξάγουν και αλλάζουν κείμενο. Το `substring(κείμενο FROM μοτίβο)` επιστρέφει το κομμάτι που ταιριάζει με την πρώτη παρένθεση του μοτίβου. Το `regexp_replace` αντικαθιστά.

```sql
SELECT sku,
       substring(sku FROM '-([0-9]+)$') AS number,
       regexp_replace(sku, '[0-9]+$', 'XXX') AS masked
FROM products
WHERE sku ~ '^(BK|LP)-';
```

ΑΠΟΤΕΛΕΣΜΑ

```
    sku     | number |   masked
------------+--------+------------
 BK-LIT-001 | 001    | BK-LIT-XXX
 BK-LIT-002 | 002    | BK-LIT-XXX
 BK-LIT-003 | 003    | BK-LIT-XXX
 BK-LIT-004 | 004    | BK-LIT-XXX
 BK-PRG-001 | 001    | BK-PRG-XXX
 BK-PRG-002 | 002    | BK-PRG-XXX
 BK-PRG-003 | 003    | BK-PRG-XXX
 BK-DB-001  | 001    | BK-DB-XXX
 BK-DB-002  | 002    | BK-DB-XXX
 BK-TEC-001 | 001    | BK-TEC-XXX
 LP-001     | 001    | LP-XXX
 LP-002     | 002    | LP-XXX
 LP-003     | 003    | LP-XXX
 LP-004     | 004    | LP-XXX
(14 rows)
```

ΠΑΓΙΔΑ

Τα regular expressions είναι ισχυρά και δύσκολα στο διάβασμα. Για ένα απλό πρόθεμα ή επίθημα το `LIKE` είναι σαφέστερο. Κράτα τα regex για επικύρωση μορφής και εξαγωγή κομματιών.

Τόνοι και unaccent

Στο κεφάλαιο 2 το `ILIKE` δεν βρήκε τα Χανιά από το `χανια`. Οι πελάτες όμως γράφουν συχνά χωρίς τόνους. Η επέκταση `unaccent` αφαιρεί τόνους και διαλυτικά. Μια επέκταση ενεργοποιείται μία φορά ανά βάση με `CREATE EXTENSION`.

```sql
CREATE EXTENSION IF NOT EXISTS unaccent;

SELECT unaccent('Καφετιέρα espresso') AS plain;

SELECT name
FROM products
WHERE unaccent(name) ILIKE '%' || unaccent('καφετιερα') || '%';
```

```
ΑΠΟΤΕΛΕΣΜΑ

CREATE EXTENSION

       plain
--------------------
 Καφετιερα espresso
(1 row)

        name
--------------------
 Καφετιέρα espresso
(1 row)
```

Η αναζήτηση `καφετιερα`, που είναι η τέταρτη πιο συχνή στα events, βρίσκει τώρα το προϊόν. Εφαρμόζεις το `unaccent` και στις δύο πλευρές, ώστε να μην έχει σημασία ποια πλευρά έχει τόνους.

Trigrams

Ένα **trigram** είναι μια ακολουθία τριών διαδοχικών χαρακτήρων. Η επέκταση `pg_trgm` σπάει κάθε κείμενο στα trigrams του και μετρά την ομοιότητα δύο κειμένων από το πόσα trigrams μοιράζονται. Δύο κείμενα με ένα ορθογραφικό λάθος έχουν τα περισσότερα trigrams κοινά.

```
SQL

CREATE EXTENSION IF NOT EXISTS pg_trgm;

SELECT show_trgm('laptop') AS trigrams;
SELECT similarity('laptop', 'labtop') AS typo, similarity('laptop', 'desktop') AS other;
```

```
ΑΠΟΤΕΛΕΣΜΑ

CREATE EXTENSION

              trigrams
-------------------------------------
 {"  l"," la",apt,lap,"op ",pto,top}
(1 row)

 typo |   other
------+------------
  0.4 | 0.15384616
(1 row)
```

Κάθε λέξη συμπληρώνεται με κενά στην αρχή και στο τέλος, γι' αυτό υπάρχουν trigrams όπως `"  l"`. Για κείμενα με ελληνικούς χαρακτήρες το `show_trgm` τυπώνει κωδικούς αντί για γράμματα, αλλά η ομοιότητα υπολογίζεται κανονικά.

Το `similarity` συγκρίνει ολόκληρα κείμενα. Όταν ψάχνεις μια λέξη μέσα σε ένα μεγαλύτερο κείμενο, όπως μια αναζήτηση μέσα σε όνομα προϊόντος, το `word_similarity` είναι καλύτερο. Μετρά πόσο μοιάζει η αναζήτηση με το πιο κοντινό κομμάτι του κειμένου.

```sql
SELECT name,
       round(similarity(lower(unaccent(name)), 'καρεκλα γραφειου')::numeric, 2) AS whole,
       round(word_similarity('καρεκλα', lower(unaccent(name)))::numeric, 2) AS word
FROM products
WHERE category_id = 13
ORDER BY word DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
            name            | whole | word
----------------------------+-------+------
 Εργονομική καρέκλα         |  0.29 | 1.00
 Γραφείο ρυθμιζόμενου ύψους |  0.22 | 0.00
 Φωτιστικό γραφείου LED     |  0.29 | 0.00
(3 rows)
```

Με το `similarity` το `Φωτιστικό γραφείου LED` βγαίνει σχεδόν ίσο με την καρέκλα, επειδή μοιράζεται τη λέξη `γραφειου`. Το `word_similarity` με τη λέξη που μετράει βρίσκει την καρέκλα με βαθμό 1.

Η επέκταση ορίζει και τελεστές. Το `κείμενο % κείμενο` είναι αληθές όταν η ομοιότητα ξεπερνά ένα όριο, από προεπιλογή 0.3. Το `<->` δίνει την απόσταση, δηλαδή ένα μείον την ομοιότητα. Είναι βολικό για `ORDER BY`. Οι τελεστές αυτοί χρησιμοποιούν GIN και GiST indexes, οπότε σε μεγάλους πίνακες η fuzzy αναζήτηση μένει γρήγορη. Το κεφάλαιο 35 το δείχνει.

```sql
SELECT name, round((lower(unaccent(name)) <-> 'ακουστικα anc')::numeric, 2) AS distance
FROM products
WHERE lower(unaccent(name)) % 'ακουστικα anc'
ORDER BY distance;
```

ΑΠΟΤΕΛΕΣΜΑ

```
     name      | distance
---------------+----------
 Ακουστικά ANC |     0.00
(1 row)
```

Full text search

Τα trigrams δεν ξέρουν τίποτα από γλώσσα. Το `καφετιέρα` και το `καφετιέρες` μοιάζουν, αλλά το `θόρυβος` και το `θορύβου` λιγότερο. Το **full text search** δουλεύει με λέξεις. Κάθε λέξη ανάγεται στη ρίζα της, το **lexeme**, με βάση τους κανόνες μιας γλώσσας. Η PostgreSQL έχει ρυθμίσεις για τα ελληνικά.

Ένα κείμενο μετατρέπεται σε `tsvector`, τη λίστα των lexemes του με τις θέσεις τους. Μια αναζήτηση μετατρέπεται σε `tsquery`. Ο τελεστής `@@` ελέγχει αν ταιριάζουν.

```sql
SELECT to_tsvector('greek', 'Ο θόρυβος του θορύβου δεν είναι θορυβώδης') AS vector,
       websearch_to_tsquery('greek', 'θόρυβοι') AS query;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                          vector                          |  query
----------------------------------------------------------+---------
 'δεν':5 'εινα':6 'θορυβ':2,4 'θορυβωδ':7 'ο':1 'τ':3 | 'θορυβ'
(1 row)
```

Τρεις μορφές της ίδιας λέξης έγιναν το ίδιο lexeme, `θορυβ`, χωρίς τόνους και με πεζά. Η ελληνική ρύθμιση δεν έχει λίστα με συνηθισμένες λέξεις που αγνοούνται, οπότε το `ο` και το `του` μένουν στο vector, το δεύτερο ως `τ`.

Το `websearch_to_tsquery` δέχεται αναζητήσεις όπως τις γράφουν οι χρήστες. Λέξεις με κενό σημαίνουν όλες, το `or` σημαίνει οποιαδήποτε, το `-` μπροστά από λέξη σημαίνει να λείπει και τα εισαγωγικά σημαίνουν ακριβή φράση.

SQL

```
SELECT r.id, r.title
FROM reviews AS r
WHERE to_tsvector('greek', r.title || ' ' || r.body)
      @@ websearch_to_tsquery('greek', 'θόρυβος');
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |      title
----+------------------
 12 | Ησυχία επιτέλους
 20 | Θορυβώδες
 28 | Αθόρυβο
(3 rows)
```

Τρεις κριτικές γράφουν για θόρυβο με τρεις διαφορετικές μορφές της λέξης. Η μία από αυτές είναι χωρίς τόνους. Το `ILIKE '%θόρυβος%'` δεν θα έβρισκε καμία.

Κατάταξη και αποσπάσματα

Το `ts_rank` δίνει βαθμό σχετικότητας, με βάση το πόσες φορές και πόσο κοντά εμφανίζονται οι λέξεις. Το `setweight` δίνει βάρος σε κομμάτια του κειμένου, ώστε μια λέξη στον τίτλο να μετρά περισσότερο από μια λέξη στο σώμα. Το `ts_headline` επιστρέφει απόσπασμα με τις λέξεις σημαδεμένες.

SQL

```
WITH docs AS (
  SELECT r.id, r.title, r.body,
         setweight(to_tsvector('greek', r.title), 'A') ||
         setweight(to_tsvector('greek', r.body), 'B') AS vector
  FROM reviews AS r
)
SELECT id, title,
       round(ts_rank(vector, q)::numeric, 3) AS rank,
       ts_headline('greek', body, q, 'StartSel=[, StopSel=]') AS snippet
FROM docs, websearch_to_tsquery('greek', 'laptop or λάπτοπ') AS q
WHERE vector @@ q
ORDER BY rank DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |       title        | rank  |                              snippet
----+--------------------+-------+-----------------------------------------------------------------
 33 | Πολύ καλό laptop   | 0.304 | Γρήγορο και αθόρυβο. Η οθόνη θα μπορούσε να είναι πιο φωτεινή.
 18 | Καθαρή εικόνα      | 0.122 | Η οθόνη 4K είναι κρυστάλλινη και το USB-C φορτίζει το [laptop].
 25 | Ελαφρύ και γρήγορο | 0.122 | Το [λάπτοπ] είναι πανάλαφρο και η μπαταρία κρατάει όλη μέρα.
 36 | Ζεσταίνεται        | 0.122 | Το [laptop] ζεσταίνεται πολύ όταν τρέχω Docker.
(4 rows)
```

Η κριτική με τη λέξη στον τίτλο είναι πρώτη. Το `websearch_to_tsquery` στο `FROM` με κόμμα υπολογίζεται μία φορά και το `q` χρησιμοποιείται σε τρία σημεία.

ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL

Σε πραγματική εφαρμογή δεν υπολογίζεις το `to_tsvector` σε κάθε αναζήτηση. Το κρατάς σε generated column τύπου `tsvector`, όπως στο κεφάλαιο 17, με GIN index από πάνω. Τότε η αναζήτηση σε εκατομμύρια κείμενα διαρκεί χιλιοστά του δευτερολέπτου.

Ποιο εργαλείο πότε

Ανάγκη	Εργαλείο
πρόθεμα ή επίθημα, απλό μοτίβο	`LIKE`, `ILIKE`
επικύρωση μορφής, εξαγωγή κομματιών	regular expressions
τόνοι	`unaccent` σε συνδυασμό με τα υπόλοιπα
ορθογραφικά λάθη, κομμάτι λέξης, διπλοεγγραφές	trigrams
λέξεις με κλίσεις, πολλές λέξεις, κατάταξη	full text search

ΚΡΑΤΑ ΑΥΤΟ

Τα regular expressions ελέγχουν και εξάγουν μοτίβα με `~`. Το `unaccent` αφαιρεί τόνους. Τα trigrams μετρούν ομοιότητα και αντέχουν λάθη, με `similarity`, `word_similarity`, `%` και `<->`. Το full text search ανάγει λέξεις σε lexemes με `to_tsvector` και ψάχνει με `@@` και `websearch_to_tsquery`.

Ασκήσεις

ΑΣΚΗΣΗ 28.1 Email με κεφαλαία

Βρες με regular expression τους πελάτες που το email τους περιέχει τουλάχιστον ένα κεφαλαίο λατινικό γράμμα. Εμφάνισε `id` και email.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |              email
----+----------------------------------
 21 | Giorgos.Papadopoulos@Example.gr
(1 row)
```

ΑΣΚΗΣΗ 28.2 Αναζήτηση χωρίς τόνους

Βρες τα προϊόντα που το όνομα ή η περιγραφή τους περιέχει τη λέξη `ηχειο` ή τη λέξη `μικροφωνο`, χωρίς να σε ενδιαφέρουν τόνοι και κεφαλαία.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      name
-----------------
 Ηχείο Bluetooth
 Μικρόφωνο USB
(2 rows)
```

ΑΣΚΗΣΗ 28.3 Από αναζήτηση σε προϊόν

Για κάθε διαφορετική αναζήτηση των events, βρες το προϊόν με τη μεγαλύτερη `word_similarity` ανάμεσα στην αναζήτηση χωρίς τόνους και σε πεζά και στο όνομα του προϊόντος χωρίς τόνους και σε πεζά. Εμφάνισε μόνο όσες έχουν βαθμό τουλάχιστον 0.5.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
         q         |          name          | score
-------------------+------------------------+-------
 ακουστικα         | Ακουστικά ANC          |  1.00
 ακουστικα anc     | Ακουστικά ANC          |  1.00
 δωροκαρτα         | Δωροκάρτα 50€          |  1.00
 καφετιερα         | Καφετιέρα espresso     |  1.00
 πληκτρολογιο      | Μηχανικό πληκτρολόγιο  |  1.00
 laptop            | Laptop Aero 14         |  1.00
 python            | Μαθαίνω Python         |  1.00
 postgres          | PostgreSQL σε βάθος    |  0.89
 οθονη 4k          | Οθόνη 27" 4K           |  0.75
 postgresql βιβλιο | PostgreSQL σε βάθος    |  0.61
 καρεκλα γραφειου  | Φωτιστικό γραφείου LED |  0.53
(11 rows)
```

ΑΣΚΗΣΗ 28.4 Διπλοεγγραφές πελατών

Βρες ζεύγη πελατών με ομοιότητα ονοματεπωνύμου τουλάχιστον 0.6, αφού μετατρέψεις τα ελληνικά γράμματα σε λατινικά με `translate(lower(unaccent(όνομα)), 'αβγδεζηθικλμνξοπρστυφχψω', 'abgdezitiklmnxoprsstyfhpo')`. Κάθε ζεύγος μία φορά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |        latin         | id |        latin         | score
----+----------------------+----+----------------------+-------
  2 | giorgos papadopoylos | 21 | giorgos papadopoulos |  0.74
(1 row)
```

ΑΣΚΗΣΗ 28.5 Κριτικές για καφέ

Με full text search, βρες τις κριτικές που αναφέρουν καφετιέρα ή espresso, ταξινομημένες κατά `ts_rank`. Εμφάνισε `id`, τίτλο και βαθμό με τρία δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |         title          | rank
----+------------------------+-------
  6 | Espresso σαν καφετέρια | 0.068
 11 | Διαρροή                | 0.030
(2 rows)
```

ΑΣΚΗΣΗ 28.6 Αποσπάσματα για ήχο

Βρες τις κριτικές που περιέχουν τη λέξη ήχος σε οποιαδήποτε μορφή και εμφάνισε `id` και απόσπασμα του σώματος με τις λέξεις ανάμεσα σε `[` και `]`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |                              snippet
----+-----------------------------------------------------------------------
  2 | Καθαρός [ήχος] στο podcast μας χωρίς επιπλέον κάρτα [ήχου].
 13 | Το ηχείο αντέχει νερό και άμμο. Δυνατός [ήχος] για το μέγεθός του.
 23 | Εξαιρετικος [ηχος] και μπαταρια που κραταει πολυ.
 37 | Ο [ήχος] καλός, η εφαρμογή του κινητού όμως είναι κακή.
(4 rows)
```

ΜΕΡΟΣ V

Απόδοση

Ένα query μπορεί να είναι σωστό και αργό. Εδώ ανοίγουμε τη μηχανή. Πώς αποθηκεύονται οι γραμμές, πώς δουλεύουν τα indexes, πώς διαβάζεις ένα query plan και τι κάνεις όταν ο planner διαλέγει λάθος.

ΚΕΦΑΛΑΙΟ 29

Πώς αποθηκεύει τα δεδομένα η PostgreSQL

Ένας πίνακας είναι ένα αρχείο χωρισμένο σε σελίδες των 8 KB. Κάθε αλλαγή μιας γραμμής γράφει νέα έκδοση της και αφήνει την παλιά στη θέση της. Το VACUUM καθαρίζει τις παλιές εκδόσεις. Αυτές οι τρεις ιδέες εξηγούν τα περισσότερα φαινόμενα απόδοσης των επόμενων κεφαλαίων.

Το εργαστήριο

Από εδώ και πέρα χρειαζόμαστε μεγαλύτερους πίνακες. Ένα query σε 162 παραγγελίες είναι γρήγορο όπως κι αν το γράψεις. Το schema `lab` έχει 200.000 πελάτες, 2.000.000 παραγγελίες, 50.000 προϊόντα και 1.000.000 events. Το φορτώνεις με `bash data/setup.sh --lab`. Χρειάζεται ένα έως δύο λεπτά.

Ένα **schema** είναι ένας χώρος ονομάτων μέσα στη βάση. Ο `lab.orders` είναι διαφορετικός πίνακας από τον `orders` που χρησιμοποιούσαμε ως τώρα, με παρόμοιες στήλες.

SQL

```
SELECT relname AS table_name,
       pg_size_pretty(pg_table_size(oid)) AS table_size,
       pg_size_pretty(pg_indexes_size(oid)) AS index_size
FROM pg_class
WHERE relnamespace = 'lab'::regnamespace AND relkind = 'r'
ORDER BY pg_table_size(oid) DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 table_name | table_size | index_size
------------+------------+------------
 orders     | 132 MB     | 43 MB
 events     | 107 MB     | 21 MB
 customers  | 25 MB      | 4408 kB
 products   | 8968 kB    | 1112 kB
(4 rows)
```

Ο `pg_class` είναι ο κατάλογος της PostgreSQL με κάθε πίνακα, index και view. Το `'lab'::regnamespace` μετατρέπει το όνομα του schema στον αριθμό του. Οι πίνακες του εργαστηρίου έχουν μόνο primary keys, οπότε τα indexes τους είναι μικρά. Στο επόμενο κεφάλαιο θα προσθέσουμε κι άλλα.

Σελίδες

Κάθε πίνακας αποθηκεύεται σε αρχεία στον δίσκο, χωρισμένα σε **σελίδες** των 8 KB. Η σελίδα είναι η μονάδα με την οποία η PostgreSQL διαβάζει και γράφει. Για να διαβάσει μία γραμμή, διαβάζει ολόκληρη τη σελίδα της. Ο κατάλογος κρατά πόσες σελίδες και πόσες γραμμές έχει περίπου κάθε πίνακας.

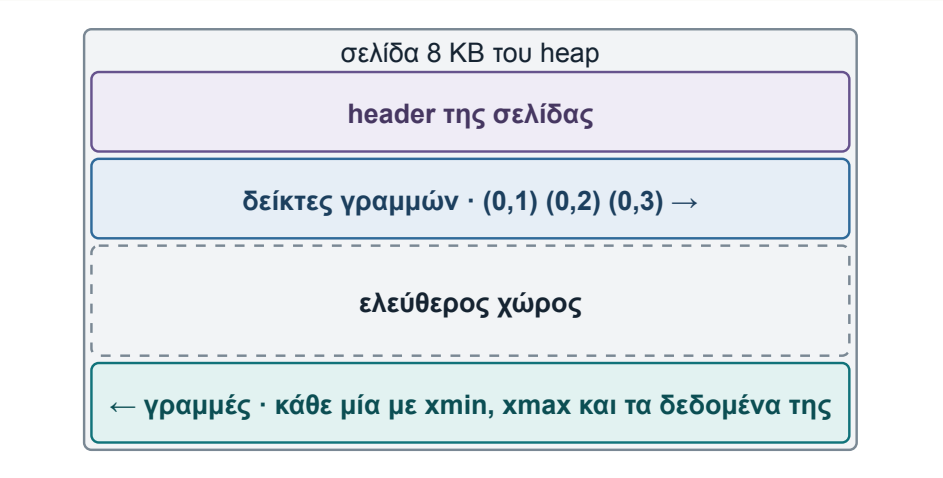
SQL

```
SELECT relname, relpages, reltuples::bigint AS rows,
       round(reltuples / relpages) AS rows_per_page
FROM pg_class
WHERE relname IN ('orders', 'customers') AND relnamespace = 'lab'::regnamespace;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  relname  | relpages |  rows   | rows_per_page
-----------+----------+---------+---------------
 customers |     3249 |  200000 |            62
 orders    |    16924 | 2000000 |           118
(2 rows)
```

Ο `lab.orders` χωρά περίπου 118 παραγγελίες σε κάθε σελίδα. Ένα query που διαβάζει ολόκληρο τον πίνακα διαβάζει όλες τις σελίδες του, είτε χρειάζεται μία γραμμή είτε όλες. Αυτό είναι το **sequential scan** και το κόστος του είναι ανάλογο του μεγέθους του πίνακα.



Σχήμα 29.1 · Μια σελίδα του heap. Οι δείκτες μεγαλώνουν από πάνω και οι γραμμές από κάτω, μέχρι να γεμίσει ο ελεύθερος χώρος.

Η δομή όπου ζουν οι γραμμές ενός πίνακα λέγεται **heap**. Οι γραμμές δεν έχουν σειρά. Μια νέα γραμμή μπαίνει σε όποια σελίδα έχει χώρο. Κάθε γραμμή έχει μια φυσική διεύθυνση, το `ctid`, που είναι ένα ζεύγος αριθμού σελίδας και θέσης μέσα στη σελίδα.

SQL

```
SELECT ctid, id, status, created_at::date
FROM lab.orders
WHERE id IN (1, 2, 118, 119, 120);
```

ΑΠΟΤΕΛΕΣΜΑ

```
  ctid   | id  |  status   | created_at
---------+-----+-----------+------------
 (0,1)   |   1 | delivered | 2021-01-01
 (0,2)   |   2 | delivered | 2021-01-01
 (0,118) | 118 | delivered | 2021-01-01
 (1,1)   | 119 | delivered | 2021-01-01
 (1,2)   | 120 | delivered | 2021-01-01
(5 rows)
```

Οι γραμμές 1 έως 118 είναι στη σελίδα 0 και η 119 ξεκινά τη σελίδα 1, επειδή ο πίνακας γράφτηκε με τη σειρά των `id` και δεν έχει αλλάξει από τότε.

Κάθε αλλαγή είναι μια νέα έκδοση

Η PostgreSQL δεν αλλάζει μια γραμμή στη θέση της. Ένα `UPDATE` γράφει μια νέα έκδοση της γραμμής και σημαδεύει την παλιά ως τελειωμένη. Αυτός ο μηχανισμός λέγεται **MVCC**, multiversion concurrency control. Επιτρέπει σε όσους διαβάζουν να βλέπουν την παλιά έκδοση όσο κάποιος γράφει τη νέα, χωρίς να περιμένει ο ένας τον άλλον. Το κεφάλαιο 37 δείχνει τις συνέπειες για τα transactions.

Κάθε έκδοση έχει δύο κρυφές στήλες. Το `xmin` είναι το transaction που την έγραψε. Το `xmax` είναι το transaction που τη διέγραψε ή την αντικατέστησε, ή 0 όσο είναι ζωντανή. Για να μην αλλάξουμε τον κοινό πίνακα, δουλεύουμε σε ένα αντίγραφο 100.000 παραγγελιών.

SQL

```
CREATE TABLE lab.orders_copy AS
SELECT * FROM lab.orders WHERE id <= 100000;

SELECT ctid, xmin, xmax, id, status FROM lab.orders_copy WHERE id = 1;
UPDATE lab.orders_copy SET status = 'refunded' WHERE id = 1;
SELECT ctid, xmin, xmax, id, status FROM lab.orders_copy WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 100000

 ctid  | xmin | xmax | id |  status
-------+------+------+----+-----------
 (0,1) | 3677 |    0 |  1 | delivered
(1 row)

UPDATE 1

  ctid   | xmin | xmax | id |  status
---------+------+------+----+----------
 (879,1) | 3681 |    0 |  1 | refunded
(1 row)
```

Η γραμμή άλλαξε `ctid`. Η νέα έκδοση γράφτηκε σε νέα σελίδα στο τέλος του πίνακα και έχει νέο `xmin`. Η παλιά έκδοση είναι ακόμη στη σελίδα 0, αόρατη για κάθε νέο transaction. Λέγεται **dead tuple**.

Dead tuples και VACUUM

Ένα `DELETE` αφήνει επίσης dead tuples και ένα `UPDATE` πολλών γραμμών αφήνει πολλά. Η επέκταση `pgstattuple` μετρά ακριβώς τι υπάρχει σε έναν πίνακα.

SQL

```
CREATE EXTENSION IF NOT EXISTS pgstattuple;

UPDATE lab.orders_copy SET total = total + 1 WHERE id % 2 = 0;

SELECT pg_size_pretty(table_len) AS size, tuple_count AS live,
       dead_tuple_count AS dead, round(free_percent::numeric, 1) AS free_pct
FROM pgstattuple('lab.orders_copy');
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

UPDATE 50000

 size  |  live  | dead  | free_pct
-------+--------+-------+----------
 10 MB | 100000 | 50001 |      0.6
(1 row)
```

Ο πίνακας έχει 100.000 ζωντανές γραμμές και 50.000 νεκρές. Η παλιά έκδοση της παραγγελίας 1 δεν μετράει πια. Η PostgreSQL καθαρίζει μόνη της νεκρές εκδόσεις μέσα σε μια σελίδα όταν τη διαβάζει και χρειάζεται χώρο, μια μικρή δουλειά που λέγεται pruning. Οι υπόλοιπες νεκρές γραμμές πιάνουν χώρο και κάθε sequential scan τις διαβάζει για να τις προσπεράσει.

Το **VACUUM** βρίσκει τις νεκρές εκδόσεις που δεν χρειάζεται πια κανένα transaction και ελευθερώνει τον χώρο τους για νέες γραμμές.

SQL

```
VACUUM lab.orders_copy;

SELECT pg_size_pretty(table_len) AS size, tuple_count AS live,
       dead_tuple_count AS dead, round(free_percent::numeric, 1) AS free_pct
FROM pgstattuple('lab.orders_copy');
```

ΑΠΟΤΕΛΕΣΜΑ

```
VACUUM

 size  |  live  | dead | free_pct
-------+--------+------+----------
 10 MB | 100000 |    0 |     31.0
(1 row)
```

Οι νεκρές γραμμές έφυγαν, αλλά το αρχείο δεν μίκρυνε. Ο χώρος έγινε ελεύθερος μέσα στις σελίδες και θα χρησιμοποιηθεί από τα επόμενα `INSERT` και `UPDATE`. Το `VACUUM FULL` ξαναγράφει τον πίνακα από την αρχή και επιστρέφει τον χώρο στο λειτουργικό, αλλά κλειδώνει τον πίνακα για όσο διαρκεί.

Στην πράξη δεν τρέχεις `VACUUM` με το χέρι. Ο **autovacuum** είναι μια διεργασία του server που παρακολουθεί πόσες γραμμές άλλαξαν σε κάθε πίνακα και τρέχει `VACUUM` και `ANALYZE` όταν χρειάζεται. Ο `pg_stat_user_tables` δείχνει πότε έτρεξε τελευταία φορά.

> **ΠΑΓΙΔΑ**
> Ένα transaction που μένει ανοιχτό για ώρες εμποδίζει το `VACUUM` να καθαρίσει οποιαδήποτε έκδοση έγινε νεκρή μετά την έναρξή του, σε όλους τους πίνακες. Οι πίνακες μεγαλώνουν χωρίς λόγο και τα queries αργούν. Αυτό είναι το **bloat** και η πιο συχνή αιτία του είναι ένα ξεχασμένο `BEGIN`. Στο `pg_stat_activity` τα βρίσκεις με `state = 'idle in transaction'`.

HOT updates και fillfactor

Κάθε νέα έκδοση γραμμής θα έπρεπε να ενημερώσει και κάθε index του πίνακα, αφού η διεύθυνσή της άλλαξε. Η PostgreSQL έχει μια βελτιστοποίηση. Αν καμία στήλη με index δεν άλλαξε και η νέα έκδοση χωρά στην ίδια σελίδα, γράφεται εκεί και τα indexes δεν αγγίζονται. Αυτό λέγεται **HOT update**, heap only tuple.

Για να υπάρχει χώρος στην ίδια σελίδα, ένας πίνακας με πολλά updates μπορεί να έχει χαμηλότερο **fillfactor**. Με `ALTER TABLE ... SET (fillfactor = 80)` τα `INSERT` γεμίζουν τις σελίδες ως το 80% και αφήνουν το υπόλοιπο για updates.

TOAST

Μια γραμμή πρέπει να χωρά σε μια σελίδα. Για μεγάλες τιμές, όπως μακριά κείμενα ή μεγάλα jsonb, η PostgreSQL χρησιμοποιεί το **TOAST**. Συμπιέζει την τιμή και, αν είναι ακόμη μεγάλη, την αποθηκεύει σε ξεχωριστό πίνακα και κρατά στη γραμμή μόνο έναν δείκτη.

SQL

```
CREATE TABLE lab.toast_demo (body text);
INSERT INTO lab.toast_demo VALUES (repeat('SQL ', 25000));

SELECT octet_length(body) AS bytes, pg_column_size(body) AS stored_bytes
FROM lab.toast_demo;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 1

 bytes  | stored_bytes
--------+--------------
 100000 |         1160
(1 row)
```

Ένα κείμενο 100.000 bytes με πολλές επαναλήψεις αποθηκεύτηκε σε λίγο πάνω από 1 KB. Το TOAST είναι αόρατο στα queries. Εξηγεί όμως γιατί ένα `SELECT *` που φέρνει μεγάλες στήλες είναι πολύ

ακριβότερο από ένα `SELECT` με λίγες μικρές στήλες. Οι μεγάλες τιμές διαβάζονται και αποσυμπιέζονται μόνο όταν τις ζητήσεις.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Οι πίνακες είναι σελίδες των 8 KB και ένα sequential scan τις διαβάζει όλες. Κάθε `UPDATE` και `DELETE` αφήνει μια νεκρή έκδοση που καθαρίζει το `VACUUM`, συνήθως μέσω του autovacuum. Τα μακροχρόνια transactions εμποδίζουν τον καθαρισμό. Οι μεγάλες τιμές μπαίνουν σε TOAST και διαβάζονται μόνο όταν τις ζητήσεις.

Ασκήσεις

ΑΣΚΗΣΗ 29.1 Μέγεθος ανά γραμμή

Για κάθε πίνακα του schema `lab` εμφάνισε όνομα, πλήθος γραμμών από τον κατάλογο και μέσο μέγεθος γραμμής σε bytes, δηλαδή μέγεθος πίνακα διά γραμμές, ως ακέραιο. Ταξινόμησε από το μεγαλύτερο μέσο μέγεθος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   relname   |  rows   | bytes_per_row
-------------+---------+---------------
 products    |   50000 |           184
 customers   |  200000 |           133
 events      | 1000000 |           112
 orders_copy |  100000 |           107
 orders      | 2000000 |            69
(5 rows)
```

ΑΣΚΗΣΗ 29.2 Η διεύθυνση μιας γραμμής

Βρες σε ποια σελίδα του `lab.orders` βρίσκονται οι παραγγελίες 500.000 και 2.000.000. Το `ctid` μετατρέπεται σε κείμενο και το πρώτο του μέρος είναι η σελίδα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   id    |    ctid    | page
---------+------------+-------
  500000 | (4231,8)   |  4231
 2000000 | (16923,28) | 16923
(2 rows)
```

ΑΣΚΗΣΗ 29.3 Διαγραφές και χώρος

Στο αντίγραφο `lab.orders_copy` της θεωρίας, διέγραψε τις ακυρωμένες παραγγελίες και μέτρησε με `pgstattuple` ζωντανές και νεκρές γραμμές. Μετά τρέξε `VACUUM` και μέτρησε ξανά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
DELETE 7070

 live  | dead
-------+------
 92930 | 7070
(1 row)

VACUUM

 live  | dead
-------+------
 92930 |    0
(1 row)
```

ΑΣΚΗΣΗ 29.4 Πόσο συμπιέζεται το JSON

Για τα events με `id` από 1 έως 5, σύγκρινε το μέγεθος του `payload` ως κείμενο με το μέγεθος που πιάνει ως `jsonb` στον δίσκο, με `octet_length` και `pg_column_size`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |                   payload                    | text_bytes | jsonb_bytes
----+----------------------------------------------+------------+-------------
  1 | {"device": "mobile", "product_id": 29433}    |         41 |          55
  2 | {"device": "mobile", "product_id": 36554}    |         41 |          55
  3 | {"device": "tablet", "product_id": 16764}    |         41 |          55
  4 | {"device": "desktop", "product_id": 49381}   |         42 |          55
  5 | {"device": "tablet", "product_id": 3389}     |         40 |          53
(5 rows)
```

ΑΣΚΗΣΗ 29.5 Fillfactor και μέγεθος

Φτιάξε δύο αντίγραφα των πρώτων 20.000 παραγγελιών του `lab.orders`, το `lab.ff100` με την προεπιλογή και το `lab.ff70` με `fillfactor` 70. Μέτρησε πόσες διαφορετικές σελίδες χρησιμοποιούν οι γραμμές καθενός, από το `ctid` όπως στην άσκηση 29.2.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 20000

CREATE TABLE

INSERT 0 20000

 table_name | used_pages
------------+------------
 ff100      |        170
 ff70       |        243
(2 rows)
```

ΚΕΦΑΛΑΙΟ 30

B-tree indexes

Ένα index είναι μια ξεχωριστή δομή, ταξινομημένη κατά μία ή περισσότερες στήλες, που δείχνει στις γραμμές του πίνακα. Με αυτό η βάση βρίσκει λίγες γραμμές ανάμεσα σε εκατομμύρια χωρίς να διαβάσει όλο τον πίνακα. Το αν θα το χρησιμοποιήσει εξαρτάται από το πώς γράφεις τη συνθήκη.

Χωρίς index

Οι παραγγελίες ενός πελάτη στον `lab.orders` δεν έχουν κανένα index που να βοηθά. Για να τις βρει, η βάση πρέπει να διαβάσει κάθε σελίδα. Το `EXPLAIN` μπροστά από ένα query δεν το εκτελεί. Τυπώνει το **plan**, το σχέδιο εκτέλεσης που διάλεξε η PostgreSQL. Θα το διαβάσουμε αναλυτικά στο επόμενο κεφάλαιο. Εδώ αρκεί να δεις τι είδους διάβασμα γίνεται. Το `COSTS OFF` κρύβει τους αριθμούς κόστους.

SQL

```
EXPLAIN (COSTS OFF)
SELECT id, total FROM lab.orders WHERE customer_id = 150000;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Gather
  Workers Planned: 2
  ->  Parallel Seq Scan on orders
        Filter: (customer_id = 150000)
```

Το `Seq Scan` διαβάζει όλο τον πίνακα και το `Filter` πετά τις γραμμές που δεν ταιριάζουν. Εδώ ο πίνακας είναι τόσο μεγάλος που η PostgreSQL μοίρασε τη δουλειά σε διεργασίες που τρέχουν παράλληλα. Αυτό είναι το `Parallel` και το `Gather`, που θα δούμε στο κεφάλαιο 33. Το `EXPLAIN ANALYZE` εκτελεί το query και προσθέτει τον πραγματικό χρόνο.

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT id, total FROM lab.orders WHERE customer_id = 150000;
```

```
ΑΠΟΤΕΛΕΣΜΑ

Gather (actual rows=10.00 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared read=16924
  ->  Parallel Seq Scan on orders (actual rows=3.33 loops=3)
        Filter: (customer_id = 150000)
        Rows Removed by Filter: 666663
        Buffers: shared read=16924
Planning Time: 0.012 ms
Execution Time: 28.485 ms
```

Η γραμμή `Rows Removed by Filter` λέει ότι για να βρει λίγες παραγγελίες η βάση εξέτασε και πέταξε περίπου δύο εκατομμύρια γραμμές. Το `Execution Time` στο τέλος είναι ο συνολικός χρόνος. Στο μηχάνημά σου θα είναι διαφορετικός.

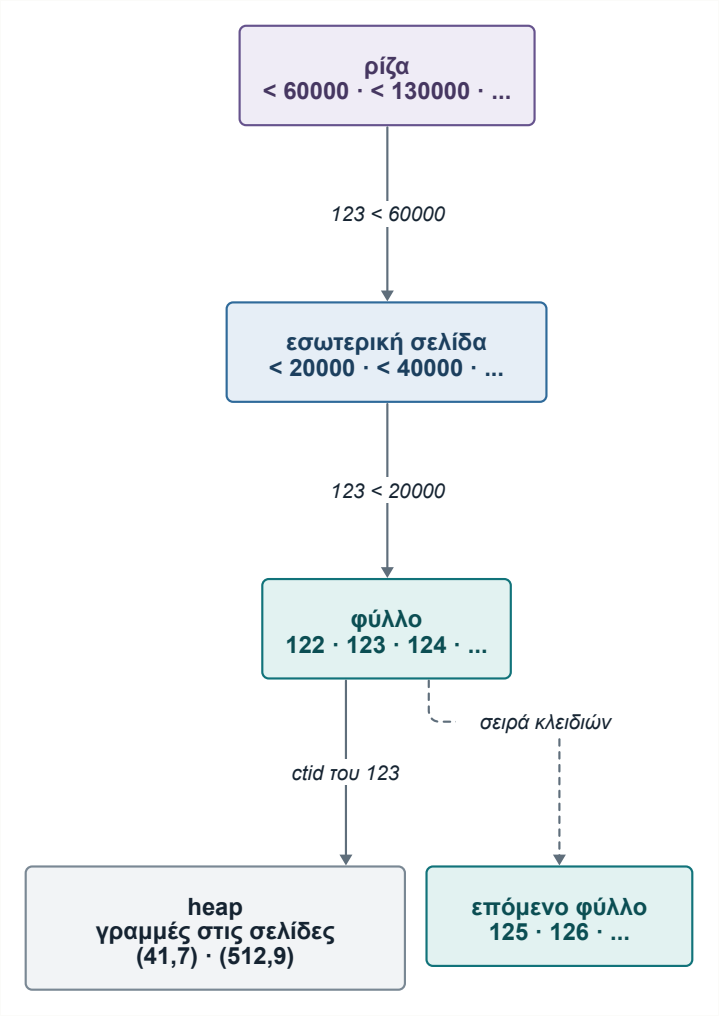
CREATE INDEX

```sql
SQL

CREATE INDEX orders_customer_idx ON lab.orders (customer_id);

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT id, total FROM lab.orders WHERE customer_id = 150000;
```

```
ΑΠΟΤΕΛΕΣΜΑ

CREATE INDEX

Bitmap Heap Scan on orders (actual rows=10.00 loops=1)
  Recheck Cond: (customer_id = 150000)
  Heap Blocks: exact=10
  Buffers: shared read=13
  ->  Bitmap Index Scan on orders_customer_idx (actual rows=10.00 loops=1)
        Index Cond: (customer_id = 150000)
        Index Searches: 1
        Buffers: shared read=3
Planning:
  Buffers: shared hit=15 read=5
Planning Time: 0.226 ms
Execution Time: 0.292 ms
```

Ο χρόνος έπεσε κατά τάξεις μεγέθους. Το plan δεν διαβάζει πια όλο τον πίνακα. Η βάση βρήκε στο index τις θέσεις των γραμμών και διάβασε μόνο τις σελίδες που τις περιέχουν. Η μορφή `Bitmap` εξηγείται λίγο παρακάτω. Τα `actual rows` έχουν δεκαδικά από την PostgreSQL 18, γιατί είναι μέσος όρος ανά εκτέλεση του κόμβου.

Ένα **B-tree** index είναι ένα ισορροπημένο δέντρο από σελίδες των 8 KB. Τα φύλλα κρατούν τις τιμές της στήλης ταξινομημένες, καθεμία με το `ctid` της γραμμής της στον πίνακα. Οι σελίδες πάνω από τα φύλλα κρατούν όρια που οδηγούν στο σωστό φύλλο. Για να βρει μια τιμή, η βάση ξεκινά από τη ρίζα και κατεβαίνει. Ακόμη και για δισεκατομμύρια γραμμές το δέντρο έχει βάθος τέσσερα ή πέντε επίπεδα.



Σχήμα 30.1 · Αναζήτηση στο B-tree. Από τη ρίζα στο σωστό φύλλο και από εκεί με το ctid στη γραμμή του πίνακα. Τα φύλλα είναι συνδεδεμένα με τη σειρά των τιμών.

Επειδή τα φύλλα είναι ταξινομημένα και συνδεδεμένα, το B-tree εξυπηρετεί και διαστήματα. Η βάση βρίσκει την αρχή και διαβάζει φύλλα προς τα δεξιά μέχρι το τέλος. Γι' αυτό ένα B-tree βοηθά τα `=`, `<`, `<=`, `>`, `>=`, `BETWEEN`, `IN`, `IS NULL` και το `ORDER BY` της ίδιας στήλης.

Bitmap scans

Η βάση δεν διάβασε το index και μετά τον πίνακα γραμμή προς γραμμή. Διάλεξε έναν τρόπο σε δύο βήματα που αξίζει να δούμε με πελάτη που έχει πολλές παραγγελίες. Στο εργαστήριο οι πελάτες με μικρό `id` έχουν πολύ περισσότερες από τους υπόλοιπους.

```sql
EXPLAIN (COSTS OFF)
SELECT id, total FROM lab.orders WHERE customer_id = 123;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Bitmap Heap Scan on orders
  Recheck Cond: (customer_id = 123)
  ->  Bitmap Index Scan on orders_customer_idx
        Index Cond: (customer_id = 123)
```

Το `Bitmap Index Scan` διαβάζει από το index όλες τις θέσεις και φτιάχνει έναν χάρτη των σελίδων που τις περιέχουν. Το `Bitmap Heap Scan` διαβάζει μετά αυτές τις σελίδες με τη σειρά τους στον δίσκο, κάθε σελίδα μία φορά. Όταν οι γραμμές είναι πολλές και σκορπισμένες, αυτό είναι φθηνότερο από το να πηγαίνει στον πίνακα για κάθε θέση χωριστά. Το `Recheck Cond` είναι ο έλεγχος της συνθήκης στις γραμμές που διαβάστηκαν.

Σύνθετα indexes

Ένα index μπορεί να έχει πολλές στήλες. Ταξινομείται κατά την πρώτη, μετά κατά τη δεύτερη μέσα σε κάθε τιμή της πρώτης, όπως ένας τηλεφωνικός κατάλογος κατά επώνυμο και όνομα. Αυτό αποφασίζει ποια queries εξυπηρετεί.

SQL

```
CREATE INDEX orders_customer_created_idx ON lab.orders (customer_id, created_at);

EXPLAIN (COSTS OFF)
SELECT id, created_at FROM lab.orders
WHERE customer_id = 123
ORDER BY created_at DESC
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Limit
  ->  Index Scan Backward using orders_customer_created_idx on orders
        Index Cond: (customer_id = 123)
```

Οι παραγγελίες του πελάτη 123 είναι συνεχόμενες στο index και ήδη ταξινομημένες κατά ημερομηνία. Η βάση διαβάζει το index ανάποδα με `Index Scan Backward` και σταματά μετά από πέντε. Δεν χρειάζεται `Sort`. Αυτό είναι το index που κάνει το `LATERAL` του κεφαλαίου 14 γρήγορο.

Ο κανόνας για τη σειρά των στηλών είναι απλός. Βάλε πρώτες τις στήλες που συγκρίνεις με ισότητα και μετά τη στήλη του διαστήματος ή του `ORDER BY`. Ένα index στο `(customer_id, created_at)` εξυπηρετεί ένα query μόνο για το `customer_id`, αλλά όχι εύκολα ένα query μόνο για το `created_at`.

ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL

Η PostgreSQL 18 έχει **skip scan**. Αν η πρώτη στήλη ενός σύνθετου index έχει λίγες διαφορετικές τιμές, η βάση μπορεί να χρησιμοποιήσει το index για συνθήκη μόνο στη δεύτερη, ψάχνοντας χωριστά μέσα σε κάθε τιμή της πρώτης. Ένα index στο `(status, created_at)`, με έξι καταστάσεις, εξυπηρετεί έτσι και ένα φίλτρο μόνο στην ημερομηνία. Σε παλαιότερες εκδόσεις θα χρειαζόταν δεύτερο index.

Συνθήκες που δεν χρησιμοποιούν index

Το index είναι ταξινομημένο κατά την τιμή της στήλης. Αν η συνθήκη εφαρμόζει μια συνάρτηση στη στήλη, η ταξινόμηση δεν βοηθά. Το κεφάλαιο 6 σε προειδοποίησε για το `created_at::date`. Ας το δούμε.

SQL

```
CREATE INDEX orders_created_idx ON lab.orders (created_at);

EXPLAIN (COSTS OFF)
SELECT count(*) FROM lab.orders WHERE created_at::date = '2026-06-15';

EXPLAIN (COSTS OFF)
SELECT count(*) FROM lab.orders
WHERE created_at >= '2026-06-15' AND created_at < '2026-06-16';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Finalize Aggregate
  ->  Gather
        Workers Planned: 2
        ->  Partial Aggregate
              ->  Parallel Seq Scan on orders
                    Filter: ((created_at)::date = '2026-06-15'::date)

Aggregate
  ->  Index Only Scan using orders_created_idx on orders
        Index Cond: ((created_at >= '2026-06-15 00:00:00+03'::timestamp with time zone)
AND (created_at < '2026-06-16 00:00:00+03'::timestamp with time zone))
```

Η πρώτη μορφή διαβάζει όλο τον πίνακα. Η δεύτερη λέει το ίδιο με όρους της στήλης και χρησιμοποιεί το index. Μια συνθήκη που μπορεί να χρησιμοποιήσει index λέγεται **sargable**. Ο γενικός κανόνας είναι να αφήνεις τη στήλη μόνη της στη μία πλευρά της σύγκρισης και να κάνεις τις πράξεις στην άλλη.

Indexes σε εκφράσεις

Κάποιες φορές η συνάρτηση είναι ουσιαστική, όπως στην αναζήτηση email χωρίς διάκριση κεφαλαίων. Τότε φτιάχνεις index πάνω στην ίδια την έκφραση.

SQL

```
CREATE INDEX customers_email_lower_idx ON lab.customers (lower(email));

EXPLAIN (COSTS OFF)
SELECT id FROM lab.customers WHERE lower(email) = 'user4242@example.gr';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Bitmap Heap Scan on customers
  Recheck Cond: (lower(email) = 'user4242@example.gr'::text)
  ->  Bitmap Index Scan on customers_email_lower_idx
        Index Cond: (lower(email) = 'user4242@example.gr'::text)
```

Το query πρέπει να γράφει την έκφραση ακριβώς όπως το index. Ένα `CREATE UNIQUE INDEX` πάνω στο `lower(email)` είναι και ο τρόπος να απαγορεύσεις δύο email που διαφέρουν μόνο στα κεφαλαία, το πρόβλημα που έμεινε ανοιχτό στο κεφάλαιο 17.

Partial indexes

Ένα index μπορεί να περιέχει μόνο τις γραμμές που περνούν ένα `WHERE`. Οι παραγγελίες σε εκκρεμότητα είναι λίγες χιλιάδες σε δύο εκατομμύρια. Είναι ακριβώς αυτές που ψάχνει κάθε λίγο μια εφαρμογή.

SQL

```
CREATE INDEX orders_pending_idx ON lab.orders (created_at) WHERE status = 'pending';

SELECT relname, pg_size_pretty(pg_relation_size(oid)) AS size
FROM pg_class
WHERE relname IN ('orders_pending_idx', 'orders_created_idx')
ORDER BY relname;

EXPLAIN (COSTS OFF)
SELECT id FROM lab.orders WHERE status = 'pending' AND created_at < '2026-06-29';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

      relname       | size
--------------------+-------
 orders_created_idx | 43 MB
 orders_pending_idx | 48 kB
(2 rows)

Index Scan using orders_pending_idx on orders
  Index Cond: (created_at < '2026-06-29 00:00:00+03'::timestamp with time zone)
```

Το partial index είναι εκατοντάδες φορές μικρότερο και η βάση το χρησιμοποιεί όταν η συνθήκη του query περιέχει τη συνθήκη του index.

Covering indexes και Index Only Scan

Ένα `Index Scan` βρίσκει τη θέση στο index και μετά διαβάζει τη γραμμή από τον πίνακα. Αν όμως όλες οι στήλες που χρειάζεται το query υπάρχουν μέσα στο index, η βάση δεν χρειάζεται τον πίνακα. Αυτό είναι το `Index Only Scan`. Με `INCLUDE` προσθέτεις σε ένα index στήλες που δεν χρησιμοποιούνται για αναζήτηση, μόνο για να διαβάζονται.

SQL

```
CREATE INDEX orders_customer_total_idx ON lab.orders (customer_id) INCLUDE (total);
VACUUM lab.orders;

EXPLAIN (COSTS OFF)
SELECT sum(total) FROM lab.orders WHERE customer_id = 123;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

VACUUM

Aggregate
  ->  Index Only Scan using orders_customer_total_idx on orders
        Index Cond: (customer_id = 123)
```

Το `VACUUM` χρειάζεται επειδή το `Index Only Scan` ελέγχει έναν χάρτη ορατότητας του πίνακα, που λέει ποιες σελίδες έχουν μόνο γραμμές ορατές σε όλους. Το κεφάλαιο 32 δείχνει τι γίνεται όταν ο χάρτης δεν είναι ενημερωμένος.

Το κόστος ενός index

Κάθε index επιταχύνει κάποια queries και επιβαρύνει κάθε `INSERT`, `UPDATE` και `DELETE`, αφού πρέπει να ενημερωθεί κι αυτό. Πιάνει χώρο στον δίσκο και στη μνήμη. Ένα index που δεν χρησιμοποιεί κανένα query είναι καθαρό κόστος.

SQL

```
SELECT indexrelname AS index_name,
       pg_size_pretty(pg_relation_size(indexrelid)) AS size,
       idx_scan
FROM pg_stat_user_indexes
WHERE relname = 'orders' AND schemaname = 'lab'
ORDER BY pg_relation_size(indexrelid) DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         index_name          | size  | idx_scan
-----------------------------+-------+----------
 orders_customer_created_idx | 60 MB |        0
 orders_customer_total_idx   | 60 MB |        0
 orders_created_idx          | 43 MB |        3
 orders_pkey                 | 43 MB |        0
 orders_customer_idx         | 18 MB |        1
 orders_pending_idx          | 48 kB |        0
(6 rows)
```

Η στήλη `idx_scan` μετρά πόσες φορές χρησιμοποιήθηκε κάθε index από την τελευταία επαναφορά των στατιστικών. Σε ένα σύστημα που τρέχει καιρό, ένα index με μηδέν είναι υποψήφιο για διαγραφή.

ΠΑΓΙΔΑ

Σε έναν πίνακα που χρησιμοποιείται, το `CREATE INDEX` κλειδώνει τις εγγραφές για όσο διαρκεί. Σε production γράφεις `CREATE INDEX CONCURRENTLY`. Χτίζει το index χωρίς να μπλοκάρει `INSERT` και `UPDATE`, με το κόστος ότι παίρνει περισσότερο χρόνο και δεν τρέχει μέσα σε transaction.

ΚΡΑΤΑ ΑΥΤΟ

Ένα B-tree κρατά τιμές ταξινομημένες και εξυπηρετεί ισότητα, διαστήματα και `ORDER BY`. Σε σύνθετα indexes βάζεις πρώτα τις στήλες ισότητας. Αφήνεις τη στήλη μόνη της στη σύγκριση, αλλιώς φτιάχνεις index στην έκφραση. Τα partial indexes καλύπτουν μικρά υποσύνολα και το `INCLUDE` επιτρέπει `Index Only Scan`. Κάθε index έχει κόστος σε κάθε εγγραφή.

Ασκήσεις

ΑΣΚΗΣΗ 30.1 Αναζήτηση προϊόντος με όνομα

Φτιάξε index στο `name` του `lab.products` και δείξε με `EXPLAIN (COSTS OFF)` ότι χρησιμοποιείται για αναζήτηση με ισότητα στο προϊόν `Οθόνη Delta ECC-103`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Index Scan using products_name_idx on products
  Index Cond: (name = 'Οθόνη Delta ECC-103'::text)
```

ΑΣΚΗΣΗ 30.2 Προθέματα και collation

Με το index της προηγούμενης άσκησης, δείξε με `EXPLAIN` ότι το `name LIKE 'Καφετιέρα Aero%'` δεν το χρησιμοποιεί. Φτιάξε δεύτερο index με `text_pattern_ops` και δείξε ότι τώρα χρησιμοποιείται.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Seq Scan on products
  Filter: (name ~~ 'Καφετιέρα Aero%'::text)

CREATE INDEX

Bitmap Heap Scan on products
  Filter: (name ~~ 'Καφετιέρα Aero%'::text)
  ->  Bitmap Index Scan on products_name_pattern_idx
        Index Cond: ((name ~>=~ 'Καφετιέρα Aero'::text) AND (name ~<~ 'Καφετιέρα Aerp'::text))
```

ΑΣΚΗΣΗ 30.3 Τα τελευταία events ενός πελάτη

Φτιάξε το index που επιτρέπει να βρίσκονται τα τρία πιο πρόσφατα events του πελάτη 777 στον `lab.events` χωρίς ταξινόμηση. Δείξε το plan.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Limit
  ->  Index Scan Backward using events_customer_time_idx on events
        Index Cond: (customer_id = 777)
```

ΑΣΚΗΣΗ 30.4 Sargable μήνας

Το query `SELECT count(*) FROM lab.events WHERE extract(year FROM occurred_at) = 2025 AND extract(month FROM occurred_at) = 3` διαβάζει όλο τον πίνακα. Φτιάξε index στο `occurred_at` και ξαναγράψε το query ώστε να τον χρησιμοποιεί. Δείξε το plan και το αποτέλεσμα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Aggregate
  ->  Index Only Scan using events_time_idx on events
        Index Cond: ((occurred_at >= '2025-03-01 00:00:00+02'::timestamp with time zone)
AND (occurred_at < '2025-04-01 00:00:00+03'::timestamp with time zone))

 count
-------
 56881
(1 row)
```

ΑΣΚΗΣΗ 30.5 Partial και unique

Στον `lab.customers`, φτιάξε unique index που εγγυάται ότι δεν υπάρχουν δύο πελάτες με το ίδιο email χωρίς διάκριση κεφαλαίων. Μετά δοκίμασε να προσθέσεις πελάτη 200001 με email `USER1@EXAMPLE.GR`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

ERROR:  duplicate key value violates unique constraint "customers_email_ci_idx"
DETAIL:  Key (lower(email))=(user1@example.gr) already exists.
```

ΑΣΚΗΣΗ 30.6 Index μόνο για ανάγνωση

Φτιάξε index που επιτρέπει `Index Only Scan` για το query που μετρά τα events ανά `kind` για τον Μάρτιο του 2025. Κάνε `VACUUM` στον πίνακα και δείξε το plan.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

VACUUM

HashAggregate
  Group Key: kind
  ->  Index Only Scan using events_time_kind_idx on events
        Index Cond: ((occurred_at >= '2025-03-01 00:00:00+02'::timestamp with time zone)
AND (occurred_at < '2025-04-01 00:00:00+03'::timestamp with time zone))
```

ΚΕΦΑΛΑΙΟ 31

Διαβάζοντας το EXPLAIN

Το `EXPLAIN` δείχνει το plan που διάλεξε ο planner για ένα query. Είναι ένα δέντρο από κόμβους. Κάθε κόμβος παίρνει γραμμές από τα παιδιά του, κάνει μια δουλειά και τις δίνει στον γονέα του. Δίπλα σε κάθε κόμβο υπάρχει η εκτίμηση του planner για το κόστος και για τις γραμμές που θα βγουν.

Πριν ξεκινήσουμε

Σε μεγάλους πίνακες η PostgreSQL μοιράζει συχνά ένα query σε πολλές διεργασίες που δουλεύουν παράλληλα, όπως είδες στο κεφάλαιο 30. Τα παράλληλα plans έχουν επιπλέον κόμβους που δυσκολεύουν την πρώτη ανάγνωση. Σε αυτό το κεφάλαιο τα απενεργοποιούμε για τη σύνδεσή μας. Το κεφάλαιο 33 τα εξηγεί.

SQL

```
SET max_parallel_workers_per_gather = 0;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET
```

Το `SET` αλλάζει μια ρύθμιση μόνο για την τρέχουσα σύνδεση. Κάθε άλλη σύνδεση και κάθε νέα σύνδεση βλέπει την κανονική τιμή.

Ένας κόμβος

SQL

```
EXPLAIN SELECT * FROM lab.customers;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on customers  (cost=0.00..5249.00 rows=200000 width=98)
```

Το plan έχει έναν κόμβο, ένα `Seq Scan`. Μέσα στην παρένθεση υπάρχουν τέσσερις αριθμοί.

Πεδίο	Σημασία
`cost=0.00..5249.00`	κόστος εκκίνησης και συνολικό κόστος
`rows=200000`	πόσες γραμμές θα βγάλει ο κόμβος κατά την εκτίμηση

Πεδίο	Σημασία
`width=98`	μέσο μέγεθος γραμμής σε bytes

Το **κόστος** δεν είναι χρόνος. Είναι μονάδες του planner. Η ανάγνωση μιας σελίδας με σειρά κοστίζει 1, η επεξεργασία μιας γραμμής 0.01. Ο `lab.customers` έχει 3.249 σελίδες και 200.000 γραμμές, οπότε το κόστος είναι 3.249 επί 1 συν 200.000 επί 0.01, δηλαδή 5.249. Μπορείς να το επαληθεύσεις.

```
SQL

SELECT relpages * current_setting('seq_page_cost')::numeric
     + reltuples * current_setting('cpu_tuple_cost')::numeric AS seq_scan_cost
FROM pg_class
WHERE oid = 'lab.customers'::regclass;
```

```
ΑΠΟΤΕΛΕΣΜΑ

 seq_scan_cost
---------------
          5249
(1 row)
```

Το **κόστος εκκίνησης** είναι η δουλειά πριν βγει η πρώτη γραμμή. Για ένα `Seq Scan` είναι μηδέν, αφού η πρώτη γραμμή βγαίνει αμέσως. Για ένα `Sort` είναι σχεδόν όλο το κόστος, αφού πρέπει να διαβάσει τα πάντα πριν δώσει την πρώτη ταξινομημένη γραμμή. Η διάκριση μετράει όταν υπάρχει `LIMIT`.

Εκτιμήσεις γραμμών

Με ένα `WHERE`, ο planner πρέπει να εκτιμήσει πόσες γραμμές θα περάσουν.

```
SQL

EXPLAIN SELECT * FROM lab.customers WHERE city = 'Κοζάνη';
```

```
ΑΠΟΤΕΛΕΣΜΑ

Seq Scan on customers  (cost=0.00..5749.00 rows=2060 width=98)
  Filter: (city = 'Κοζάνη'::text)
```

Το κόστος μεγάλωσε λίγο, αφού κάθε γραμμή ελέγχεται με το φίλτρο. Οι εκτιμώμενες γραμμές έπεσαν σε λίγες χιλιάδες. Ο planner δεν μέτρησε. Διάβασε από τα **statistics** του πίνακα ότι η Κοζάνη είναι περίπου το 1% των πελατών. Η ακρίβεια αυτών των εκτιμήσεων είναι το θέμα του κεφαλαίου 34. Ό,τι αποφασίζει ο planner βασίζεται σε αυτές.

Ένα δέντρο

Ένα πραγματικό query έχει πολλούς κόμβους. Οι τρεις πόλεις με τις περισσότερες παραγγελίες τον Ιούνιο του 2026 δίνουν ένα plan με επτά.

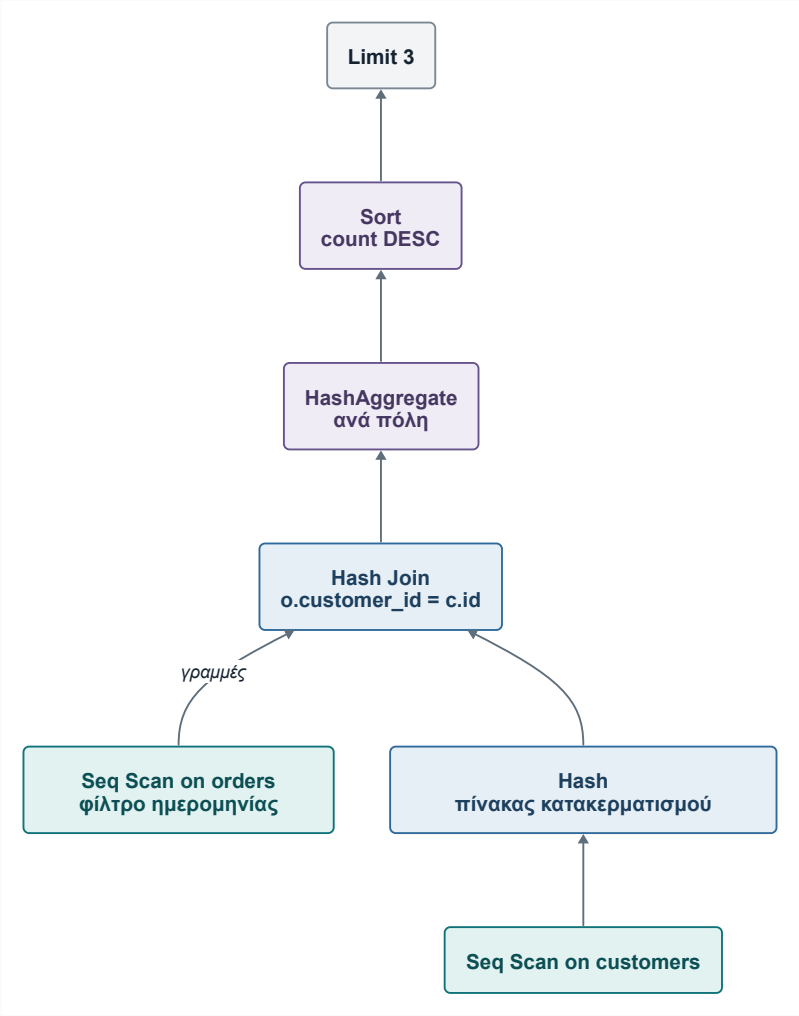
SQL

```
EXPLAIN
SELECT c.city, count(*)
FROM lab.orders AS o
JOIN lab.customers AS c ON c.id = o.customer_id
WHERE o.created_at >= '2026-06-01'
GROUP BY c.city
ORDER BY count(*) DESC
LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Limit  (cost=52482.62..52482.63 rows=3 width=22)
  ->  Sort  (cost=52482.62..52482.65 rows=11 width=22)
        Sort Key: (count(*)) DESC
        ->  HashAggregate  (cost=52482.37..52482.48 rows=11 width=22)
              Group Key: c.city
              ->  Hash Join  (cost=8921.00..52331.96 rows=30081 width=14)
                    Hash Cond: (o.customer_id = c.id)
                    ->  Seq Scan on orders o  (cost=0.00..41924.00 rows=30081 width=4)
                          Filter: (created_at >= '2026-06-01 00:00:00+03'::timestamp with time zone)
                    ->  Hash  (cost=5249.00..5249.00 rows=200000 width=18)
                          ->  Seq Scan on customers c  (cost=0.00..5249.00 rows=200000 width=18)
```

Κάθε κόμβος γράφεται με `->` και πιο μέσα από τον γονέα του. Οι κόμβοι στο ίδιο βάθος κάτω από έναν γονέα είναι τα παιδιά του. Το plan εκτελείται από κάτω προς τα πάνω. Τα φύλλα διαβάζουν πίνακες και οι ενδιάμεσοι κόμβοι συνδυάζουν και μετασχηματίζουν. Η ρίζα δίνει το αποτέλεσμα.



Σχήμα 31.1 · Το ίδιο plan ως δέντρο. Οι γραμμές ρέουν από τα φύλλα προς τη ρίζα.

Διαβασμένο από κάτω προς τα πάνω, το plan λέει τα εξής βήματα. Διάβασε όλους τους πελάτες και φτιάξε με αυτούς έναν πίνακα κατακερματισμού στη μνήμη, τον `Hash`. Διάβασε τις παραγγελίες, κράτα όσες είναι του Ιουνίου και για καθεμία βρες τον πελάτη της στον πίνακα κατακερματισμού, στο `Hash Join`. Ομαδοποίησε ανά πόλη με `HashAggregate`. Ταξινόμησε κατά πλήθος και κράτα τρεις.

Το κόστος κάθε κόμβου περιλαμβάνει και τα παιδιά του. Το συνολικό κόστος του `Limit` στη ρίζα είναι το κόστος όλου του query. Για να βρεις πού πηγαίνει η δουλειά, κοιτάζεις πού μεγαλώνει απότομα το κόστος από παιδί σε γονέα.

Οι συνηθισμένοι κόμβοι

Κόμβος	Τι κάνει
`Seq Scan`	διαβάζει όλον τον πίνακα
`Index Scan`	βρίσκει γραμμές στο index και τις διαβάζει από τον πίνακα

Κόμβος	Τι κάνει
`Index Only Scan`	παίρνει τα πάντα από το index χωρίς τον πίνακα
`Bitmap Heap Scan`	διαβάζει με τη σειρά του δίσκου τις σελίδες που βρήκε ένα `Bitmap Index Scan`
`Sort`	ταξινομεί
`Limit`	σταματά μετά από N γραμμές
`HashAggregate`, `GroupAggregate`	`GROUP BY` με πίνακα κατακερματισμού ή πάνω σε ταξινομημένη είσοδο
`Nested Loop`, `Hash Join`, `Merge Join`	οι τρεις τρόποι να γίνει ένα join, στο κεφάλαιο 33
`Materialize`	κρατά γραμμές στη μνήμη για να τις ξαναδιαβάσει
`CTE Scan`	διαβάζει ένα CTE που υπολογίστηκε χωριστά

Το LIMIT αλλάζει το plan

Χωρίς `LIMIT`, ο planner βελτιστοποιεί το συνολικό κόστος. Με `LIMIT`, το κόστος εκκίνησης γίνεται σημαντικό, γιατί το query θα σταματήσει νωρίς. Οι δέκα μεγαλύτερες παραγγελίες είναι ένα καλό παράδειγμα.

SQL

```
EXPLAIN SELECT id, total FROM lab.orders ORDER BY total DESC LIMIT 10;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Limit  (cost=80143.28..80143.31 rows=10 width=14)
  ->  Sort  (cost=80143.28..85143.28 rows=2000000 width=14)
        Sort Key: total DESC
        ->  Seq Scan on orders  (cost=0.00..36924.00 rows=2000000 width=14)
```

Δεν υπάρχει index στο `total`, οπότε ο `Sort` πρέπει να δει όλες τις γραμμές. Το `Limit` όμως του λέει ότι αρκούν δέκα. Η PostgreSQL δεν ταξινομεί τότε δύο εκατομμύρια γραμμές. Κρατά μόνο τις δέκα καλύτερες καθώς διαβάζει, με αλγόριθμο **top N heapsort**. Αυτό θα το δεις στο `EXPLAIN ANALYZE` του επόμενου κεφαλαίου.

CTE στο plan

Στο κεφάλαιο 21 είδες ότι ένα CTE που χρησιμοποιείται μία φορά ενσωματώνεται στο κύριο query. Το plan το δείχνει. Με `MATERIALIZED`, το CTE υπολογίζεται χωριστά και το φίλτρο του κύριου query δεν μπορεί να μπει μέσα του.

```sql
EXPLAIN (COSTS OFF)
WITH o AS (SELECT * FROM lab.orders)
SELECT * FROM o WHERE id = 42;

EXPLAIN (COSTS OFF)
WITH o AS MATERIALIZED (SELECT * FROM lab.orders)
SELECT * FROM o WHERE id = 42;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Index Scan using orders_pkey on orders
  Index Cond: (id = 42)

CTE Scan on o
  Filter: (id = 42)
  CTE o
    ->  Seq Scan on orders
```

Στην πρώτη μορφή το `id = 42` έφτασε ως το index του primary key. Στη δεύτερη η βάση θα διάβαζε όλον τον πίνακα, θα τον κρατούσε ως αποτέλεσμα του CTE και μετά θα έψαχνε μέσα του μία γραμμή.

Επιλογές του EXPLAIN

Το `EXPLAIN` δέχεται επιλογές σε παρένθεση. Το `COSTS OFF` κρύβει τα κόστη. Το `VERBOSE` δείχνει τις στήλες που βγάζει κάθε κόμβος, με το όνομα του schema μπροστά από κάθε πίνακα. Το `FORMAT JSON` δίνει το plan σε μορφή κατάλληλη για εργαλεία οπτικοποίησης. Το `ANALYZE` εκτελεί το query και προσθέτει τους πραγματικούς αριθμούς, το θέμα του επόμενου κεφαλαίου.

```sql
EXPLAIN (VERBOSE, COSTS OFF)
SELECT c.first_name, o.total
FROM lab.orders AS o
JOIN lab.customers AS c ON c.id = o.customer_id
WHERE o.id = 42;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Nested Loop
  Output: c.first_name, o.total
  Inner Unique: true
  ->  Index Scan using orders_pkey on lab.orders o
        Output: o.id, o.customer_id, o.status, o.total, o.created_at, o.note
        Index Cond: (o.id = 42)
  ->  Index Scan using customers_pkey on lab.customers c
        Output: c.id, c.first_name, c.last_name, c.email, c.city, c.region, c.created_at
        Index Cond: (c.id = o.customer_id)
Query Identifier: -3485696273204513076
```

Για πειράματα υπάρχουν οι ρυθμίσεις `enable_*`, όπως η `enable_seqscan`. Αν την κάνεις `off`, ο planner αποφεύγει το `Seq Scan` όπου μπορεί, ώστε να δεις πόσο θα κόστιζε η εναλλακτική. Δεν είναι εργαλείο για production. Είναι τρόπος να ρωτήσεις τον planner γιατί δεν διάλεξε κάτι.

ΚΡΑΤΑ ΑΥΤΟ

Το plan είναι δέντρο που εκτελείται από τα φύλλα προς τη ρίζα. Κάθε κόμβος έχει κόστος εκκίνησης, συνολικό κόστος, εκτιμώμενες γραμμές και πλάτος. Το κόστος είναι σε μονάδες του planner και περιλαμβάνει τα παιδιά του κόμβου. Οι εκτιμήσεις γραμμών έρχονται από τα statistics και καθορίζουν κάθε επιλογή.

Ασκήσεις

Σε όλες τις ασκήσεις ισχύει ακόμη το `max_parallel_workers_per_gather = 0` της θεωρίας.

ΑΣΚΗΣΗ 31.1 Υπολόγισε το κόστος

Υπολόγισε από τον κατάλογο το κόστος ενός `Seq Scan` στον `lab.products` και σύγκρινέ το με το `EXPLAIN SELECT * FROM lab.products`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 computed
----------
     1616
(1 row)

Seq Scan on products  (cost=0.00..1616.00 rows=50000 width=147)
```

ΑΣΚΗΣΗ 31.2 Κόστος του φίλτρου

Δείξε με `EXPLAIN` το plan του `SELECT * FROM lab.products WHERE price > 800`. Υπολόγισε τη διαφορά του συνολικού κόστους από το plan της προηγούμενης άσκησης και εξήγησε από πού έρχεται.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on products  (cost=0.00..1741.00 rows=2997 width=147)
  Filter: (price > '800'::numeric)
```

ΑΣΚΗΣΗ 31.3 Αν δεν υπήρχε το Seq Scan

Σε transaction, κάνε `SET LOCAL enable_seqscan = off` και δείξε το plan του `SELECT sum(total) FROM lab.orders`. Σύγκρινε με το κανονικό plan και εξήγησε τι σημαίνει η νέα γραμμή που εμφανίζεται.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate  (cost=41924.00..41924.01 rows=1 width=32)
  ->  Seq Scan on orders  (cost=0.00..36924.00 rows=2000000 width=6)

SET

Aggregate  (cost=41924.00..41924.01 rows=1 width=32)
  ->  Seq Scan on orders  (cost=0.00..36924.00 rows=2000000 width=6)
        Disabled: true
```

ΑΣΚΗΣΗ 31.4 Το plan ενός anti join

Δείξε με `EXPLAIN (COSTS OFF)` το plan του query που βρίσκει τα προϊόντα του `lab.products` χωρίς κανένα event, γραμμένο με `NOT EXISTS` πάνω στο `(payload ->> 'product_id')::integer`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Hash Right Anti Join
  Hash Cond: (((e.payload ->> 'product_id'::text))::integer = p.id)
  ->  Seq Scan on events e
  ->  Hash
        ->  Index Only Scan using products_pkey on products p
```

ΑΣΚΗΣΗ 31.5 Ποιες στήλες κουβαλά κάθε κόμβος

Με `EXPLAIN (VERBOSE, COSTS OFF)`, δείξε το plan του `SELECT city, count(*) FROM lab.customers GROUP BY city`. Εντόπισε ποιες στήλες διαβάζει το `Seq Scan`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
HashAggregate
  Output: city, count(*)
  Group Key: customers.city
  ->  Seq Scan on lab.customers
        Output: id, first_name, last_name, email, city, region, created_at
Query Identifier: 4132498954773243860
```

ΚΕΦΑΛΑΙΟ 32

EXPLAIN ANALYZE και BUFFERS

Το `EXPLAIN` λέει τι σκοπεύει να κάνει ο planner. Το `EXPLAIN ANALYZE` εκτελεί το query και λέει τι έγινε πραγματικά. Η σύγκριση των δύο είναι το πιο χρήσιμο εργαλείο για ένα αργό query. Εκεί όπου η εκτίμηση απέχει πολύ από την πραγματικότητα, συνήθως βρίσκεται και το πρόβλημα.

Πραγματικοί αριθμοί

Όπως στο προηγούμενο κεφάλαιο, κλείνουμε τον παραλληλισμό για να είναι τα plans απλά.

SQL

```
SET max_parallel_workers_per_gather = 0;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET
```

SQL

```
EXPLAIN ANALYZE
SELECT id, total FROM lab.orders ORDER BY total DESC LIMIT 10;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Limit  (cost=80143.28..80143.31 rows=10 width=14) (actual time=236.554..236.555 rows=10.00 loops=1)
  Buffers: shared hit=3 read=16924
  ->  Sort  (cost=80143.28..85143.28 rows=2000000 width=14) (actual time=236.553..236.553 rows=10.00 loops=1)
        Sort Key: total DESC
        Sort Method: top-N heapsort  Memory: 25kB
        Buffers: shared hit=3 read=16924
        ->  Seq Scan on orders  (cost=0.00..36924.00 rows=2000000 width=14) (actual time=0.073..103.629
rows=2000000.00 loops=1)
              Buffers: shared read=16924
Planning:
  Buffers: shared hit=57 read=22 dirtied=2
Planning Time: 0.556 ms
Execution Time: 236.568 ms
```

Δίπλα στην εκτίμηση κάθε κόμβου υπάρχει τώρα μια δεύτερη παρένθεση με τις πραγματικές τιμές.

Πεδίο	Σημασία
`actual time=0.2..104.2`	ms μέχρι την πρώτη γραμμή και μέχρι την τελευταία
`rows=2000000.00`	γραμμές που βγήκαν, κατά μέσο όρο ανά εκτέλεση του κόμβου

Πεδίο	Σημασία
`loops=1`	πόσες φορές εκτελέστηκε ο κόμβος
`Buffers`	σελίδες που διαβάστηκαν, στο επόμενο τμήμα

Ο χρόνος κάθε κόμβου περιλαμβάνει τον χρόνο των παιδιών του. Εδώ το `Seq Scan` διάβασε δύο εκατομμύρια γραμμές σε κάτι λιγότερο από τον μισό συνολικό χρόνο και ο `Sort` χρειάστηκε τον υπόλοιπο. Η γραμμή `Sort Method` επιβεβαιώνει ότι έγινε **top N heapsort** με ελάχιστη μνήμη, όπως προέβλεψε το προηγούμενο κεφάλαιο.

Το `Planning Time` είναι ο χρόνος του planner. Το `Execution Time` είναι ο χρόνος εκτέλεσης χωρίς την αποστολή των γραμμών στον client. Και οι δύο θα διαφέρουν στο μηχάνημά σου και από εκτέλεση σε εκτέλεση.

> **ΠΑΓΙΔΑ**
> Το `EXPLAIN ANALYZE` εκτελεί πραγματικά το query. Ένα `EXPLAIN ANALYZE DELETE` διαγράφει γραμμές. Για εντολές που αλλάζουν δεδομένα, γράψε το μέσα σε `BEGIN` και `ROLLBACK`.

Loops

Όταν ένας κόμβος εκτελείται πολλές φορές, οι αριθμοί του είναι μέσοι όροι ανά εκτέλεση. Για τα σύνολα πολλαπλασιάζεις με το `loops`. Αυτό συμβαίνει κυρίως στο εσωτερικό σκέλος ενός `Nested Loop`, που εκτελείται μία φορά για κάθε γραμμή του εξωτερικού.

SQL

```
CREATE INDEX orders_customer_idx ON lab.orders (customer_id);

EXPLAIN (ANALYZE, COSTS OFF)
SELECT c.id, count(o.id)
FROM lab.customers AS c
JOIN lab.orders AS o ON o.customer_id = c.id
WHERE c.id BETWEEN 100000 AND 100020
GROUP BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

GroupAggregate (actual time=0.086..1.727 rows=21.00 loops=1)
  Group Key: c.id
  Buffers: shared hit=68 read=156
  ->  Nested Loop (actual time=0.038..1.697 rows=157.00 loops=1)
        Buffers: shared hit=68 read=156
        ->  Index Only Scan using customers_pkey on customers c (actual time=0.007..0.013 rows=21.00 loops=1)
              Index Cond: ((id >= 100000) AND (id <= 100020))
              Heap Fetches: 0
              Index Searches: 1
              Buffers: shared hit=3 read=1
        ->  Bitmap Heap Scan on orders o (actual time=0.023..0.077 rows=7.48 loops=21)
              Recheck Cond: (c.id = customer_id)
              Heap Blocks: exact=157
              Buffers: shared hit=65 read=155
              ->  Bitmap Index Scan on orders_customer_idx (actual time=0.002..0.002 rows=7.48 loops=21)
                    Index Cond: (customer_id = c.id)
                    Index Searches: 21
                    Buffers: shared hit=61 read=2
Planning:
  Buffers: shared hit=66 read=18 dirtied=3
```

```
Planning Time: 0.469 ms
Execution Time: 1.789 ms
```

Το index scan στον `lab.orders` εκτελέστηκε μία φορά για κάθε πελάτη. Τα `rows` του είναι οι παραγγελίες ενός πελάτη κατά μέσο όρο και ο χρόνος του είναι επίσης ανά εκτέλεση. Ο συνολικός χρόνος του κόμβου είναι ο χρόνος επί τα loops.

Για ακριβέστερες μετρήσεις υπάρχει το `TIMING OFF`. Η μέτρηση χρόνου σε κάθε γραμμή κάθε κόμβου έχει κόστος, που σε queries με εκατομμύρια γραμμές παραμορφώνει το αποτέλεσμα. Με `TIMING OFF` κρατάς τις γραμμές και τον συνολικό χρόνο χωρίς αυτό το κόστος.

Buffers

Από την PostgreSQL 18 το `EXPLAIN ANALYZE` δείχνει από προεπιλογή τα buffers, δηλαδή πόσες σελίδες των 8 KB χρειάστηκε κάθε κόμβος.

Όρος	Σημασία
`shared hit`	η σελίδα ήταν ήδη στη μνήμη της PostgreSQL
`shared read`	η σελίδα ήρθε από το λειτουργικό, από την cache του ή από τον δίσκο
`shared dirtied`, `written`	σελίδες που άλλαξαν ή γράφτηκαν
`temp read`, `written`	προσωρινά αρχεία, όταν η μνήμη δεν έφτασε

Οι σελίδες είναι πιο σταθερό μέτρο από τον χρόνο. Ο χρόνος εξαρτάται από το μηχάνημα και από ό,τι άλλο τρέχει. Ο αριθμός των σελίδων εξαρτάται μόνο από το plan και τα δεδομένα. Αν ένα query διαβάζει 16.000 σελίδες για να βρει δέκα γραμμές, ξέρεις πού να ψάξεις.

Η ίδια εκτέλεση δύο φορές δείχνει τη διαφορά ανάμεσα σε κρύα και ζεστή μνήμη.

```sql
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT count(*) FROM lab.events WHERE kind = 'checkout';

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT count(*) FROM lab.events WHERE kind = 'checkout';
```

```
ΑΠΟΤΕΛΕΣΜΑ

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared read=13659 dirtied=13659 written=2208
  ->  Seq Scan on events (actual rows=142762.00 loops=1)
        Filter: (kind = 'checkout'::text)
        Rows Removed by Filter: 857238
        Buffers: shared read=13659 dirtied=13659 written=2208
Planning:
  Buffers: shared hit=26 read=4 dirtied=2
Planning Time: 0.142 ms
Execution Time: 83.769 ms

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=7499 read=6160 written=59
  ->  Seq Scan on events (actual rows=142762.00 loops=1)
        Filter: (kind = 'checkout'::text)
        Rows Removed by Filter: 857238
        Buffers: shared hit=7499 read=6160 written=59
Planning Time: 0.043 ms
Execution Time: 40.525 ms
```

Την πρώτη φορά οι σελίδες ήρθαν ως `read`. Τη δεύτερη όσες χώρεσαν στη μνήμη ήταν `hit`. Η μνήμη της PostgreSQL, το `shared_buffers`, είναι 128 MB στην προεπιλογή, οπότε ένας πίνακας 107 MB χωρά σχεδόν ολόκληρος. Ένας μεγαλύτερος θα έδιωχνε σελίδες του εαυτού του.

Μνήμη και προσωρινά αρχεία

Ταξινομήσεις, hash joins και hash aggregates χρησιμοποιούν μνήμη μέχρι το όριο της ρύθμισης `work_mem`, από προεπιλογή 4 MB. Αν δεν τους φτάνει, γράφουν σε προσωρινά αρχεία στον δίσκο.

```
SQL

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT last_name, first_name FROM lab.customers ORDER BY last_name, first_name;
```

```
ΑΠΟΤΕΛΕΣΜΑ

Sort (actual rows=200000.00 loops=1)
  Sort Key: last_name, first_name
  Sort Method: external merge  Disk: 8496kB
  Buffers: shared hit=3 read=3249 written=752, temp read=1062 written=1065
  ->  Seq Scan on customers (actual rows=200000.00 loops=1)
        Buffers: shared read=3249 written=752
Planning:
  Buffers: shared hit=6 read=7 dirtied=1
Planning Time: 0.127 ms
Execution Time: 162.490 ms
```

Το `Sort Method: external merge Disk` λέει ότι η ταξινόμηση έγινε σε κομμάτια στον δίσκο και μετά συγχωνεύτηκε. Τα `temp read` και `temp written` είναι οι σελίδες των προσωρινών αρχείων. Με περισσότερη μνήμη η ίδια ταξινόμηση γίνεται εξ ολοκλήρου στη μνήμη.

SQL

```
SET work_mem = '64MB';

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT last_name, first_name FROM lab.customers ORDER BY last_name, first_name;

RESET work_mem;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET

Sort (actual rows=200000.00 loops=1)
  Sort Key: last_name, first_name
  Sort Method: quicksort  Memory: 16339kB
  Buffers: shared hit=3249
  ->  Seq Scan on customers (actual rows=200000.00 loops=1)
        Buffers: shared hit=3249
Planning Time: 0.022 ms
Execution Time: 130.672 ms

RESET
```

ΠΑΓΙΔΑ

Το `work_mem` ισχύει για κάθε κόμβο ταξινόμησης ή hash σε κάθε query κάθε σύνδεσης. Ένα query με τρεις ταξινομήσεις σε εκατό συνδέσεις μπορεί να χρησιμοποιήσει τριακόσιες φορές το `work_mem`. Αύξησέ το για συγκεκριμένα queries ή συνδέσεις με `SET`, όχι για όλο τον server χωρίς υπολογισμό.

Heap Fetches

Στο κεφάλαιο 30 το `Index Only Scan` χρειάστηκε `VACUUM`. Το `EXPLAIN ANALYZE` δείχνει γιατί. Η βάση πρέπει να ξέρει ότι μια γραμμή είναι ορατή στο transaction που διαβάζει. Αυτή η πληροφορία βρίσκεται στον πίνακα, όχι στο index. Ο **visibility map** σημαδεύει τις σελίδες όπου όλες οι γραμμές είναι ορατές σε όλους. Για αυτές το `Index Only Scan` δεν επισκέπτεται τον πίνακα. Για τις υπόλοιπες κάνει ένα **heap fetch**.

SQL

```
CREATE INDEX orders_created_total_idx ON lab.orders (created_at) INCLUDE (total);
VACUUM lab.orders;
UPDATE lab.orders SET note = 'επείγον' WHERE id % 1000 = 0 AND created_at >= '2026-06-01';

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT sum(total) FROM lab.orders WHERE created_at >= '2026-06-01';

VACUUM lab.orders;

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT sum(total) FROM lab.orders WHERE created_at >= '2026-06-01';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX
```

```
VACUUM

UPDATE 30

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=205
  ->  Index Only Scan using orders_created_total_idx on orders (actual rows=29960.00 loops=1)
        Index Cond: (created_at >= '2026-06-01 00:00:00+03'::timestamp with time zone)
        Heap Fetches: 3494
        Index Searches: 1
        Buffers: shared hit=205
Planning:
  Buffers: shared hit=10 read=4
Planning Time: 0.059 ms
Execution Time: 1.857 ms

VACUUM

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=147
  ->  Index Only Scan using orders_created_total_idx on orders (actual rows=29960.00 loops=1)
        Index Cond: (created_at >= '2026-06-01 00:00:00+03'::timestamp with time zone)
        Heap Fetches: 3437
        Index Searches: 1
        Buffers: shared hit=147
Planning:
  Buffers: shared hit=8
Planning Time: 0.173 ms
Execution Time: 1.874 ms
```

Μετά το `UPDATE`, οι σελίδες που άλλαξαν δεν ήταν πια όλες ορατές και το `Heap Fetches` έδειξε χιλιάδες επισκέψεις στον πίνακα, για όλες τις γραμμές αυτών των σελίδων. Το `VACUUM` όμως δεν το διόρθωσε. Οι νεκρές γραμμές ήταν μόνο 30 σε δύο εκατομμύρια και το `VACUUM` αποφάσισε ότι δεν αξίζει να καθαρίσει τα indexes για τόσο λίγες. Όσο τα indexes δείχνουν ακόμη στις νεκρές θέσεις, οι σελίδες τους δεν μπορούν να σημαδευτούν ως ορατές. Η επιλογή `INDEX_CLEANUP ON` αναγκάζει τον πλήρη καθαρισμό.

SQL

```
VACUUM (INDEX_CLEANUP ON) lab.orders;

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT sum(total) FROM lab.orders WHERE created_at >= '2026-06-01';
```

ΑΠΟΤΕΛΕΣΜΑ

```
VACUUM

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=118
  ->  Index Only Scan using orders_created_total_idx on orders (actual rows=29960.00 loops=1)
        Index Cond: (created_at >= '2026-06-01 00:00:00+03'::timestamp with time zone)
        Heap Fetches: 0
        Index Searches: 1
        Buffers: shared hit=118
Planning:
  Buffers: shared hit=8
Planning Time: 0.096 ms
Execution Time: 1.726 ms
```

Τώρα το `Heap Fetches` είναι μηδέν. Ένας πίνακας με συνεχείς αλλαγές και αργό autovacuum χάνει το όφελος του `Index Only Scan` χωρίς να αλλάξει το plan.

Εκτίμηση και πραγματικότητα

Η πιο σημαντική σύγκριση είναι ανάμεσα στο `rows` της εκτίμησης και στο `rows` της εκτέλεσης. Ο planner διαλέγει joins, indexes και σειρά εκτέλεσης με βάση την εκτίμηση. Αν απέχει πολύ, η επιλογή μπορεί να είναι λάθος.

SQL

```
EXPLAIN ANALYZE
SELECT * FROM lab.customers WHERE city = 'Χανιά' AND region = 'Κρήτη';
```

ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on customers  (cost=0.00..6249.00 rows=924 width=98) (actual time=0.072..13.781 rows=7991.00 loops=1)
  Filter: ((city = 'Χανιά'::text) AND (region = 'Κρήτη'::text))
  Rows Removed by Filter: 192009
  Buffers: shared hit=594 read=2655
Planning:
  Buffers: shared hit=11 read=1 dirtied=1
Planning Time: 0.159 ms
Execution Time: 13.942 ms
```

Ο planner περίμενε κάτω από χίλιες γραμμές και βρήκε σχεδόν οκτώ χιλιάδες, οκτώ φορές περισσότερες. Πολλαπλασίασε την πιθανότητα της πόλης με την πιθανότητα της περιφέρειας, σαν να ήταν ανεξάρτητες. Τα Χανιά όμως είναι πάντα στην Κρήτη. Εδώ το λάθος δεν πειράζει, αφού υπάρχει ένα μόνο πιθανό plan. Σε ένα join όμως θα μπορούσε να οδηγήσει σε `Nested Loop` αντί για `Hash Join`. Το κεφάλαιο 34 δείχνει πώς το διορθώνεις.

ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL

Για να δεις plans από την πραγματική κίνηση και όχι μόνο όσα τρέχεις με το χέρι, υπάρχει το module `auto_explain`. Καταγράφει στο log το plan κάθε query που ξεπερνά έναν χρόνο που ορίζεις, με τους πραγματικούς αριθμούς.

ΚΡΑΤΑ ΑΥΤΟ

Το `EXPLAIN ANALYZE` εκτελεί το query και δείχνει πραγματικό χρόνο, γραμμές και loops. Οι αριθμοί ενός κόμβου είναι ανά loop. Τα buffers μετρούν σελίδες και είναι το πιο σταθερό μέτρο κόστους. Ταξινομήσεις που δεν χωρούν στο `work_mem` γράφουν στον δίσκο. Μεγάλη απόκλιση εκτίμησης και πραγματικότητας είναι το πρώτο που ψάχνεις.

Ασκήσεις

Στις ασκήσεις ισχύει ακόμη το `max_parallel_workers_per_gather = 0` και το index `orders_customer_idx` της θεωρίας.

ΑΣΚΗΣΗ 32.1 Πόσες σελίδες για λίγες γραμμές

Με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)`, δείξε το plan για τα events του πελάτη 4242. Φτιάξε index στο `customer_id` του `lab.events` και δείξε το plan ξανά. Σύγκρινε τις σελίδες.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on events (actual rows=11.00 loops=1)
  Filter: (customer_id = 4242)
  Rows Removed by Filter: 999989
  Buffers: shared read=13659
Planning:
  Buffers: shared hit=5 read=1 dirtied=1
Planning Time: 0.127 ms
Execution Time: 34.468 ms

CREATE INDEX

Bitmap Heap Scan on events (actual rows=11.00 loops=1)
  Recheck Cond: (customer_id = 4242)
  Heap Blocks: exact=11
  Buffers: shared read=14
  ->  Bitmap Index Scan on events_customer_idx (actual rows=11.00 loops=1)
        Index Cond: (customer_id = 4242)
        Index Searches: 1
        Buffers: shared read=3
Planning:
  Buffers: shared hit=4 read=1
Planning Time: 0.120 ms
Execution Time: 0.138 ms
```

ΑΣΚΗΣΗ 32.2 Υπολόγισε από τα loops

Με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)`, δείξε το plan για τις παραγγελίες των πελατών 150000 έως 150100, με join ανάμεσα στον `lab.customers` και στον `lab.orders`. Από τα `rows` και τα `loops` του εσωτερικού κόμβου υπολόγισε πόσες παραγγελίες βρέθηκαν συνολικά και έλεγξέ το με `count(*)`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Nested Loop (actual rows=586.00 loops=1)
  Buffers: shared hit=320 read=574
  ->  Index Only Scan using customers_pkey on customers c (actual rows=101.00 loops=1)
        Index Cond: ((id >= 150000) AND (id <= 150100))
        Heap Fetches: 0
        Index Searches: 1
        Buffers: shared hit=3 read=2
  ->  Index Scan using orders_customer_idx on orders o (actual rows=5.80 loops=101)
        Index Cond: (customer_id = c.id)
        Index Searches: 101
        Buffers: shared hit=317 read=572
Planning:
  Buffers: shared hit=20 read=10
Planning Time: 0.245 ms
Execution Time: 1.934 ms

 count
-------
   586
(1 row)
```

ΑΣΚΗΣΗ 32.3 Αρκετή μνήμη για ένα hash

Δείξε με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)` το plan του `SELECT customer_id, count(*) FROM lab.events GROUP BY customer_id`. Μετά, μέσα στο ίδιο transaction, κάνε `SET LOCAL work_mem = '64MB'` και δείξε το ξανά. Βρες τι άλλαξε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
HashAggregate (actual rows=198658.00 loops=1)
  Group Key: customer_id
  Batches: 5  Memory Usage: 8249kB  Disk Usage: 11424kB
  Buffers: shared hit=11 read=13648, temp read=1210 written=2325
  ->  Seq Scan on events (actual rows=1000000.00 loops=1)
        Buffers: shared hit=11 read=13648
Planning:
  Buffers: shared hit=8 dirtied=2
Planning Time: 0.055 ms
Execution Time: 237.733 ms

SET

HashAggregate (actual rows=198658.00 loops=1)
  Group Key: customer_id
  Batches: 1  Memory Usage: 12313kB
  Buffers: shared read=13659
  ->  Seq Scan on events (actual rows=1000000.00 loops=1)
        Buffers: shared read=13659
Planning Time: 0.075 ms
Execution Time: 146.586 ms
```

ΑΣΚΗΣΗ 32.4 Μια εκτίμηση που αστοχεί

Με `EXPLAIN ANALYZE`, σύγκρινε την εκτίμηση και την πραγματικότητα για τους πελάτες με `city = 'Αθήνα' AND region = 'Κρήτη'`, ένα συνδυασμό που δεν υπάρχει.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on customers  (cost=0.00..6249.00 rows=9544 width=98) (actual time=13.977..13.977 rows=0.00 loops=1)
  Filter: ((city = 'Αθήνα'::text) AND (region = 'Κρήτη'::text))
  Rows Removed by Filter: 200000
  Buffers: shared read=3249 written=174
Planning Time: 0.100 ms
Execution Time: 13.984 ms
```

ΑΣΚΗΣΗ 32.5 EXPLAIN ANALYZE χωρίς συνέπειες

Μέσα σε transaction, δείξε το `EXPLAIN ANALYZE` ενός `UPDATE` που κάνει `delivered` τις παραγγελίες `shipped` του `lab.orders`. Μετά το `ROLLBACK` έλεγξε ότι το πλήθος των `shipped` δεν άλλαξε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

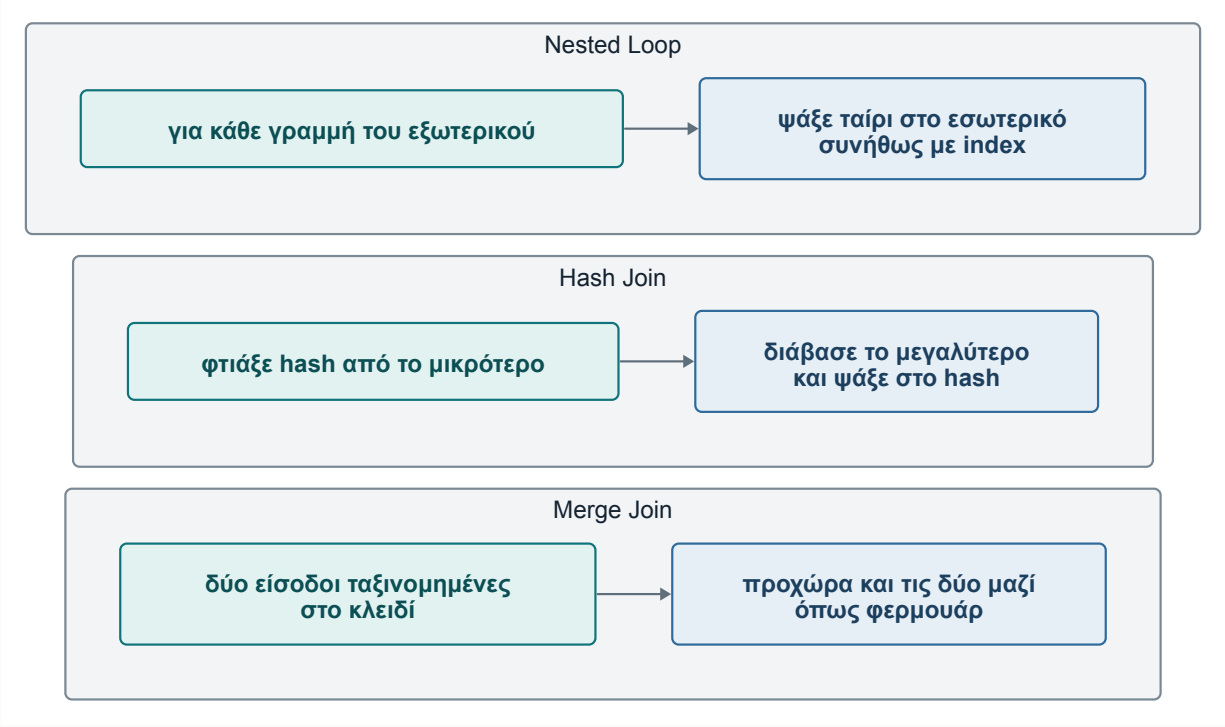
```
BEGIN

Update on orders (actual rows=0.00 loops=1)
  Buffers: shared hit=17573 read=17997 dirtied=1135 written=37
  ->  Seq Scan on orders (actual rows=1330.00 loops=1)
        Filter: (status = 'shipped'::text)
        Rows Removed by Filter: 1998670
        Buffers: shared read=16924
Planning:
  Buffers: shared read=3 dirtied=1
Planning Time: 0.222 ms
Execution Time: 72.770 ms

ROLLBACK

 shipped
---------
    1330
(1 row)
```

ΚΕΦΑΛΑΙΟ 33

Joins, aggregates και parallel query στον planner

Ένα join γράφεται με έναν τρόπο αλλά εκτελείται με τρεις. Το nested loop, το hash join και το merge join ταιριάζουν σε διαφορετικά μεγέθη και σε διαφορετική διαθέσιμη ταξινόμηση. Τα aggregates έχουν δύο αντίστοιχους αλγόριθμους. Όταν ο πίνακας είναι μεγάλος, η δουλειά μοιράζεται σε πολλές διεργασίες.

Τρεις αλγόριθμοι join



Σχήμα 33.1 · Οι τρεις τρόποι να εκτελεστεί ένα join. Το αποτέλεσμα είναι ίδιο, το κόστος εξαρτάται από τα μεγέθη και από την ταξινόμηση που υπάρχει ήδη.

Όπως στα δύο προηγούμενα κεφάλαια, ξεκινάμε χωρίς παραλληλισμό. Χρειαζόμαστε επίσης το index στο `customer_id` του `lab.orders`.

```sql
SET max_parallel_workers_per_gather = 0;
CREATE INDEX orders_customer_idx ON lab.orders (customer_id);
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET

CREATE INDEX
```

Nested Loop

Για κάθε γραμμή της εξωτερικής εισόδου, το **nested loop** ψάχνει ταίρι στην εσωτερική. Αν η εσωτερική έχει index στο κλειδί του join, κάθε αναζήτηση είναι φθηνή. Είναι ο καλύτερος αλγόριθμος όταν η εξωτερική είσοδος έχει λίγες γραμμές.

```sql
EXPLAIN (COSTS OFF)
SELECT c.first_name, o.total
FROM lab.customers AS c
JOIN lab.orders AS o ON o.customer_id = c.id
WHERE c.id BETWEEN 150000 AND 150020;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Nested Loop
  ->  Index Scan using customers_pkey on customers c
        Index Cond: ((id >= 150000) AND (id <= 150020))
  ->  Bitmap Heap Scan on orders o
        Recheck Cond: (c.id = customer_id)
        ->  Bitmap Index Scan on orders_customer_idx
              Index Cond: (customer_id = c.id)
```

Είκοσι ένας πελάτες, είκοσι μία αναζητήσεις στο index. Το κόστος μεγαλώνει γραμμικά με τις εξωτερικές γραμμές. Για χιλιάδες πελάτες θα ήταν χιλιάδες αναζητήσεις, οπότε ο planner θα διάλεγε κάτι άλλο. Χωρίς index στην εσωτερική πλευρά, κάθε αναζήτηση θα ήταν πλήρες scan και το nested loop θα χρησιμοποιούνταν μόνο για πολύ μικρούς πίνακες.

Hash Join

Το **hash join** διαβάζει πρώτα τη μικρότερη είσοδο και φτιάχνει από αυτήν πίνακα κατακερματισμού στη μνήμη. Μετά διαβάζει τη μεγαλύτερη μία φορά και για κάθε γραμμή βρίσκει το ταίρι της στον πίνακα σε σταθερό χρόνο. Δουλεύει μόνο για συνθήκες ισότητας.

```sql
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT c.city, sum(o.total)
FROM lab.customers AS c
JOIN lab.orders AS o ON o.customer_id = c.id
GROUP BY c.city;
```

ΑΠΟΤΕΛΕΣΜΑ

```
HashAggregate (actual rows=11.00 loops=1)
  Group Key: c.city
  Batches: 1  Memory Usage: 32kB
  Buffers: shared read=20173, temp read=4336 written=4336
  ->  Hash Join (actual rows=2000000.00 loops=1)
        Hash Cond: (o.customer_id = c.id)
        Buffers: shared read=20173, temp read=4336 written=4336
        ->  Seq Scan on orders o (actual rows=2000000.00 loops=1)
              Buffers: shared read=16924
        ->  Hash (actual rows=200000.00 loops=1)
              Buckets: 131072  Batches: 2  Memory Usage: 6073kB
              Buffers: shared read=3249, temp written=483
              ->  Seq Scan on customers c (actual rows=200000.00 loops=1)
                    Buffers: shared read=3249
Planning:
  Buffers: shared hit=38 read=5
Planning Time: 0.082 ms
Execution Time: 743.757 ms
```

Ο κόμβος `Hash` είναι το χτίσιμο του πίνακα από τους πελάτες. Το `Buckets` και το `Memory Usage` λένε πόσο μεγάλος βγήκε. Το `Batches: 2` λέει ότι δεν χώρεσε ολόκληρος στα 4 MB του `work_mem`. Η βάση χώρισε τότε και τις δύο εισόδους σε δύο κομμάτια, κράτησε το ένα στη μνήμη, έγραψε το άλλο σε προσωρινά αρχεία και τα ένωσε το ένα μετά το άλλο. Τα `temp` των buffers είναι αυτά τα αρχεία. Το hash join είναι ο συνηθισμένος αλγόριθμος για joins μεγάλων πινάκων χωρίς χρήσιμα indexes.

Merge Join

Το **merge join** χρειάζεται και τις δύο εισόδους ταξινομημένες στο κλειδί. Τις διαβάζει παράλληλα όπως κλείνει ένα φερμουάρ και βγάζει τα ζεύγη που ταιριάζουν. Είναι φθηνό όταν η ταξινόμηση υπάρχει ήδη, από index ή από προηγούμενο βήμα. Αν πρέπει να ταξινομηθούν μεγάλες είσοδοι μόνο για αυτό, το κόστος της ταξινόμησης συνήθως το αποκλείει.

Μπορείς να δεις τι θα κόστιζε αν απαγορεύσεις το hash join.

```sql
EXPLAIN
SELECT c.city, sum(o.total)
FROM lab.customers AS c
JOIN lab.orders AS o ON o.customer_id = c.id
GROUP BY c.city;

BEGIN;
SET LOCAL enable_hashjoin = off;

EXPLAIN
SELECT c.city, sum(o.total)
FROM lab.customers AS c
JOIN lab.orders AS o ON o.customer_id = c.id
GROUP BY c.city;

ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ

```
HashAggregate  (cost=81799.11..81799.25 rows=11 width=46)
  Group Key: c.city
  ->  Hash Join  (cost=8921.00..71799.11 rows=2000000 width=20)
        Hash Cond: (o.customer_id = c.id)
        ->  Seq Scan on orders o  (cost=0.00..36924.00 rows=2000000 width=10)
        ->  Hash  (cost=5249.00..5249.00 rows=200000 width=18)
              ->  Seq Scan on customers c  (cost=0.00..5249.00 rows=200000 width=18)

BEGIN

SET

HashAggregate  (cost=150723.71..150723.85 rows=11 width=46)
  Group Key: c.city
  ->  Merge Join  (cost=401.36..140723.71 rows=2000000 width=20)
        Merge Cond: (c.id = o.customer_id)
        ->  Index Scan using customers_pkey on customers c  (cost=0.42..8456.42 rows=200000 width=18)
        ->  Index Scan using orders_customer_idx on orders o  (cost=0.43..106767.52 rows=2000000 width=10)

ROLLBACK
```

Ο planner διάλεξε merge join. Οι πελάτες ήρθαν ταξινομημένοι από το primary key και οι παραγγελίες από το index στο `customer_id`. Το συνολικό κόστος όμως είναι μεγαλύτερο από του hash join, γι’ αυτό δεν το διάλεγε κανονικά. Το index διαβάζει τις δύο εκατομμύρια παραγγελίες με τη σειρά των πελατών, όχι με τη σειρά του δίσκου, κάτι που κοστίζει.

> **ΣΗΜΕΙΩΣΗ**
> Οι ρυθμίσεις `enable_*` δεν είναι τρόπος βελτιστοποίησης. Είναι τρόπος να δεις τις εναλλακτικές που απέρριψε ο planner και πόσο τις κοστολόγησε. Αν μια εναλλακτική είναι πραγματικά γρηγορότερη στο `EXPLAIN ANALYZE` αλλά κοστολογείται ακριβότερη, κάτι λείπει από τα statistics ή από τις σταθερές κόστους. Αυτό είναι το σωστό σημείο να ψάξεις.

Semi joins και anti joins

Το `EXISTS` και το `NOT EXISTS` του κεφαλαίου 13 δεν εκτελούνται ως subqueries γραμμή προς γραμμή. Ο planner τα μετατρέπει σε **semi join** και **anti join**, παραλλαγές των ίδιων τριών αλγορίθμων. Ένα semi join σταματά στο πρώτο ταίρι κάθε γραμμής, ένα anti join κρατά μόνο τις γραμμές χωρίς ταίρι.

```sql
EXPLAIN (COSTS OFF)
SELECT count(*)
FROM lab.customers AS c
WHERE NOT EXISTS (SELECT 1 FROM lab.orders AS o
                  WHERE o.customer_id = c.id
                    AND o.created_at >= '2026-01-01');
```

ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate
  ->  Hash Right Anti Join
        Hash Cond: (o.customer_id = c.id)
        ->  Seq Scan on orders o
              Filter: (created_at >= '2026-01-01 00:00:00+02'::timestamp with time zone)
        ->  Hash
              ->  Index Only Scan using customers_pkey on customers c
```

Aggregates

Το `GROUP BY` έχει δύο αλγορίθμους. Το **HashAggregate** κρατά έναν πίνακα κατακερματισμού με μία εγγραφή ανά ομάδα και ενημερώνει την εγγραφή σε κάθε γραμμή. Δεν χρειάζεται ταξινόμηση, αλλά χρειάζεται μνήμη ανάλογη με τον αριθμό των ομάδων. Το **GroupAggregate** διαβάζει γραμμές ήδη ταξινομημένες κατά το κλειδί και τελειώνει κάθε ομάδα μόλις αλλάξει η τιμή. Χρειάζεται ελάχιστη μνήμη, αλλά θέλει ταξινομημένη είσοδο.

```sql
EXPLAIN (COSTS OFF)
SELECT customer_id, count(*) FROM lab.orders
WHERE customer_id < 1000
GROUP BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
GroupAggregate
  Group Key: customer_id
  ->  Index Only Scan using orders_customer_idx on orders
        Index Cond: (customer_id < 1000)
```

Εδώ το index στο `customer_id` δίνει τις παραγγελίες ήδη ταξινομημένες, οπότε το `GroupAggregate` δεν χρειάζεται `Sort`. Χωρίς το `WHERE`, για όλους τους πελάτες, ο planner θα διάλεγε ανάμεσα σε `HashAggregate` με πλήρες scan και `GroupAggregate` με το index, ανάλογα με το κόστος.

Parallel query

Όταν ένα query διαβάζει μεγάλο πίνακα, η PostgreSQL μπορεί να μοιράσει τη δουλειά σε **parallel workers**, βοηθητικές διεργασίες που τρέχουν στους άλλους πυρήνες. Επαναφέρουμε τη ρύθμιση για να το δούμε.

SQL

```
RESET max_parallel_workers_per_gather;

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, BUFFERS OFF)
SELECT status, count(*) FROM lab.orders GROUP BY status;
```

ΑΠΟΤΕΛΕΣΜΑ

```
RESET

Finalize GroupAggregate (actual rows=6.00 loops=1)
  Group Key: status
  ->  Gather Merge (actual rows=18.00 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Sort (actual rows=6.00 loops=3)
              Sort Key: status
              Sort Method: quicksort  Memory: 25kB
              Worker 0:  Sort Method: quicksort  Memory: 25kB
              Worker 1:  Sort Method: quicksort  Memory: 25kB
              ->  Partial HashAggregate (actual rows=6.00 loops=3)
                    Group Key: status
                    Batches: 1  Memory Usage: 32kB
                    Worker 0:  Batches: 1  Memory Usage: 32kB
                    Worker 1:  Batches: 1  Memory Usage: 32kB
                    ->  Parallel Seq Scan on orders (actual rows=666666.67 loops=3)
Planning Time: 0.025 ms
Execution Time: 54.988 ms
```

Διάβασε το plan από κάτω. Το `Parallel Seq Scan` εκτελέστηκε σε τρεις διεργασίες, `loops=3`, δύο workers και τη βασική διεργασία. Κάθε μία διάβασε ένα τρίτο των σελίδων. Το `Partial HashAggregate` έκανε σε κάθε διεργασία ένα μερικό `GROUP BY` με έξι ομάδες. Το `Gather Merge` μάζεψε τα μερικά αποτελέσματα, 18 γραμμές. Το `Finalize GroupAggregate` τα ένωσε σε έξι.

Δεν παραλληλίζεται κάθε query. Ο planner παραλληλίζει όταν ο πίνακας ξεπερνά το μέγεθος `min_parallel_table_scan_size` και όταν το κέρδος ξεπερνά το κόστος εκκίνησης των workers. Δεν παραλληλίζει queries που αλλάζουν δεδομένα, ούτε όσα καλούν συναρτήσεις δηλωμένες ως μη ασφαλείς για παραλληλισμό. Ο αριθμός των workers ανά κόμβο ορίζεται από το `max_parallel_workers_per_gather`.

ΠΑΓΙΔΑ

Ο παραλληλισμός μειώνει τον χρόνο ενός query, όχι τη συνολική δουλειά. Τρεις διεργασίες που διαβάζουν έναν πίνακα χρησιμοποιούν τρεις πυρήνες. Σε έναν server με πολλές ταυτόχρονες συνδέσεις, ο παραλληλισμός μπορεί να κάνει πιο αργά τα υπόλοιπα queries.

ΚΡΑΤΑ ΑΥΤΟ

Το nested loop είναι για λίγες εξωτερικές γραμμές με index στην εσωτερική πλευρά. Το hash join είναι για μεγάλες εισόδους με ισότητα. Το merge join είναι για εισόδους ήδη ταξινομημένες. Το `EXISTS` γίνεται semi join και το `NOT EXISTS` anti join. Το `HashAggregate` θέλει μνήμη, το `GroupAggregate` ταξινόμηση. Ο παραλληλισμός μοιράζει ένα scan σε workers και ενώνει μερικά αποτελέσματα.

Ασκήσεις

Στις ασκήσεις ο παραλληλισμός είναι ξανά ενεργός, όπως τον άφησε η θεωρία. Υπάρχει επίσης το index `orders_customer_idx`.

ΑΣΚΗΣΗ 33.1 Από nested loop σε hash join

Με `EXPLAIN (COSTS OFF)`, δείξε το plan για τις παραγγελίες των πελατών με `id` από 150000 έως 150010 και μετά από 1 έως 50000. Εξήγησε γιατί άλλαξε ο αλγόριθμος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Nested Loop
  ->  Index Only Scan using customers_pkey on customers c
        Index Cond: ((id >= 150000) AND (id <= 150010))
  ->  Bitmap Heap Scan on orders o
        Recheck Cond: (c.id = customer_id)
        ->  Bitmap Index Scan on orders_customer_idx
              Index Cond: (customer_id = c.id)

Hash Join
  Hash Cond: (o.customer_id = c.id)
  ->  Seq Scan on orders o
  ->  Hash
        ->  Index Only Scan using customers_pkey on customers c
              Index Cond: ((id >= 1) AND (id <= 50000))
```

ΑΣΚΗΣΗ 33.2 Batches σε hash join

Μέσα σε transaction με `SET LOCAL work_mem = '1MB'` και χωρίς παραλληλισμό, δείξε με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)` το join όλων των παραγγελιών με όλους τους πελάτες που μετρά παραγγελίες ανά περιφέρεια. Εντόπισε το πλήθος των batches.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SET

SET

HashAggregate (actual rows=9.00 loops=1)
  Group Key: c.region
  Batches: 1  Memory Usage: 32kB
  Buffers: shared read=20173 written=1963, temp read=6021 written=6021
  ->  Hash Join (actual rows=2000000.00 loops=1)
        Hash Cond: (o.customer_id = c.id)
        Buffers: shared read=20173 written=1963, temp read=6021 written=6021
        ->  Seq Scan on orders o (actual rows=2000000.00 loops=1)
              Buffers: shared read=16924 written=36
        ->  Hash (actual rows=200000.00 loops=1)
              Buckets: 32768  Batches: 8  Memory Usage: 1674kB
              Buffers: shared read=3249 written=1927, temp written=962
              ->  Seq Scan on customers c (actual rows=200000.00 loops=1)
                    Buffers: shared read=3249 written=1927
Planning:
  Buffers: shared hit=18 read=1 dirtied=1 written=1
Planning Time: 0.083 ms
Execution Time: 570.532 ms
```

ΑΣΚΗΣΗ 33.3 Semi join

Με `EXPLAIN (COSTS OFF)` χωρίς παραλληλισμό, δείξε το plan για το πλήθος των πελατών του `lab.customers` που έχουν τουλάχιστον μία παραγγελία πάνω από 1400 ευρώ, με `EXISTS`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SET

Aggregate
  ->  Hash Join
        Hash Cond: (c.id = o.customer_id)
        ->  Index Only Scan using customers_pkey on customers c
        ->  Hash
              ->  HashAggregate
                    Group Key: o.customer_id
                    ->  Seq Scan on orders o
                          Filter: (total > '1400'::numeric)
```

ΑΣΚΗΣΗ 33.4 Merge join από δύο indexes

Μέσα σε transaction, απενεργοποίησε το hash join και τον παραλληλισμό και δείξε με `EXPLAIN (COSTS OFF)` το plan για το άθροισμα των παραγγελιών ανά περιφέρεια. Εντόπισε από πού έρχεται η ταξινόμηση κάθε εισόδου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SET

SET

HashAggregate
  Group Key: c.region
  ->  Merge Join
        Merge Cond: (c.id = o.customer_id)
        ->  Index Scan using customers_pkey on customers c
        ->  Index Scan using orders_customer_idx on orders o
```

ΑΣΚΗΣΗ 33.5 Πόσοι workers

Με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, BUFFERS OFF)`, δείξε το plan για το άθροισμα όλων των `total` του `lab.orders`, πρώτα με την προεπιλογή και μετά σε transaction με `SET LOCAL max_parallel_workers_per_gather = 4`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Finalize Aggregate (actual rows=1.00 loops=1)
  ->  Gather (actual rows=3.00 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate (actual rows=1.00 loops=3)
              ->  Parallel Seq Scan on orders (actual rows=666666.67 loops=3)
Planning Time: 0.030 ms
Execution Time: 39.636 ms

SET

Finalize Aggregate (actual rows=1.00 loops=1)
  ->  Gather (actual rows=4.00 loops=1)
        Workers Planned: 3
        Workers Launched: 3
        ->  Partial Aggregate (actual rows=1.00 loops=4)
              ->  Parallel Seq Scan on orders (actual rows=500000.00 loops=4)
Planning Time: 0.029 ms
Execution Time: 31.050 ms
```

ΚΕΦΑΛΑΙΟ 34

Statistics και εκτιμήσεις

Ο planner δεν μετράει γραμμές πριν αποφασίσει. Εκτιμά από statistics, μια σύνοψη κάθε στήλης που φτιάχνει το `ANALYZE` από δείγμα του πίνακα. Όταν η σύνοψη είναι ακριβής, τα plans είναι καλά. Όταν λείπει κάτι, όπως η σχέση ανάμεσα σε δύο στήλες, οι εκτιμήσεις αστοχούν και τα plans μαζί τους.

Τι κρατά το ANALYZE

Όπως στα προηγούμενα κεφάλαια, τα plans είναι χωρίς παραλληλισμό.

SQL

```
SET max_parallel_workers_per_gather = 0;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET
```

Το `ANALYZE` διαβάζει ένα τυχαίο δείγμα 30.000 γραμμών με τις προεπιλογές και γράφει για κάθε στήλη μια σύνοψη στον κατάλογο `pg_statistic`. Το view `pg_stats` την παρουσιάζει σε αναγνώσιμη μορφή. Ο autovacuum τρέχει `ANALYZE` μόνος του όταν αλλάξει αρκετό μέρος ενός πίνακα.

SQL

```
SELECT attname, null_frac, n_distinct, round(correlation::numeric, 3) AS correlation
FROM pg_stats
WHERE schemaname = 'lab' AND tablename = 'orders'
ORDER BY attname;
```

ΑΠΟΤΕΛΕΣΜΑ

```
   attname   | null_frac | n_distinct | correlation
-------------+-----------+------------+-------------
 created_at  |         0 |         -1 |       1.000
 customer_id |         0 |     108806 |      -0.004
 id          |         0 |         -1 |       1.000
 note        |    0.9806 |          1 |       1.000
 status      |         0 |          6 |       0.819
 total       |         0 |      54171 |       0.006
(6 rows)
```

Το `null_frac` είναι το ποσοστό των `NULL`. Σχεδόν όλες οι σημειώσεις λείπουν. Το `n_distinct` είναι το πλήθος των διαφορετικών τιμών. Όταν είναι αρνητικό, είναι ποσοστό των γραμμών. Το `-1` σημαίνει ότι κάθε τιμή είναι μοναδική. Το `correlation` λέει πόσο ακολουθεί η φυσική σειρά των γραμμών στον δίσκο

τη σειρά των τιμών. Κοντά στο 1 σημαίνει ότι ένα index scan στη στήλη διαβάζει σελίδες σχεδόν με τη σειρά τους. Αυτή ήταν η αιτία της έκπληξης στην άσκηση 30.6, όπου ένα απλό index στο `created_at` νίκησε ένα covering index.

Συχνές τιμές και ιστόγραμμα

Για κάθε στήλη το `ANALYZE` κρατά και δύο κατανομές. Τη λίστα των πιο συχνών τιμών με τις συχνότητές τους και ένα ιστόγραμμα για τις υπόλοιπες.

SQL

```
SELECT t.value, round(t.freq::numeric, 4) AS freq
FROM pg_stats AS s,
     unnest(s.most_common_vals::text::text[], s.most_common_freqs) AS t(value, freq)
WHERE s.schemaname = 'lab' AND s.tablename = 'customers' AND s.attname = 'city';
```

ΑΠΟΤΕΛΕΣΜΑ

```
    value    |  freq
-------------+--------
 Αθήνα       | 0.4005
 Θεσσαλονίκη | 0.1793
 Πάτρα       | 0.0824
 Ηράκλειο    | 0.0804
 Λάρισα      | 0.0615
 Βόλος       | 0.0593
 Ιωάννινα    | 0.0394
 Χανιά       | 0.0388
 Καλαμάτα    | 0.0297
 Ρόδος       | 0.0185
 Κοζάνη      | 0.0103
(11 rows)
```

Οι στήλες `most_common_vals` και `most_common_freqs` είναι arrays. Το `unnest` του κεφαλαίου 27 με δύο arrays τα ανοίγει παράλληλα σε ζεύγη τιμής και συχνότητας. Το `::text::text[]` χρειάζεται επειδή η στήλη των τιμών έχει ειδικό τύπο που χωρά τιμές οποιουδήποτε τύπου.

Η στήλη `city` έχει μόνο έντεκα τιμές, οπότε όλες χωρούν στη λίστα. Για το `city = 'Κοζάνη'` ο planner παίρνει τη συχνότητα της Κοζάνης, περίπου 1%. Την πολλαπλασιάζει με τις γραμμές του πίνακα. Αυτή ήταν η εκτίμηση του κεφαλαίου 31.

Σε στήλες με πολλές τιμές, η λίστα κρατά τις πιο συχνές και το ιστόγραμμα χωρίζει τις υπόλοιπες σε κάδους με ίσο πλήθος γραμμών. Για μια σύγκριση διαστήματος, όπως `total > 1000`, ο planner μετρά πόσοι κάδοι είναι πάνω από το 1000.

Η λίστα είναι κρίσιμη όταν η κατανομή είναι ανισόρροπη. Στο εργαστήριο οι πελάτες με μικρό `id` έχουν πολύ περισσότερες παραγγελίες.

SQL

```
EXPLAIN SELECT id FROM lab.orders WHERE customer_id = 1;
SELECT count(*) FROM lab.orders WHERE customer_id = 1;

EXPLAIN SELECT id FROM lab.orders WHERE customer_id = 150000;
SELECT count(*) FROM lab.orders WHERE customer_id = 150000;
```

```
ΑΠΟΤΕΛΕΣΜΑ

Seq Scan on orders  (cost=0.00..41924.00 rows=3600 width=8)
  Filter: (customer_id = 1)

 count
-------
  4460
(1 row)

Seq Scan on orders  (cost=0.00..41924.00 rows=18 width=8)
  Filter: (customer_id = 150000)

 count
-------
    10
(1 row)
```

Ο πελάτης 1 είναι στη λίστα των συχνών τιμών και η εκτίμηση πλησιάζει την πραγματικότητα. Ο 150000 δεν είναι. Για αυτόν ο planner υποθέτει ότι όλες οι τιμές εκτός λίστας μοιράζονται εξίσου τις υπόλοιπες γραμμές και πάλι πέφτει κοντά. Οι εκτιμήσεις σου μπορεί να διαφέρουν λίγο από του βιβλίου, αφού το δείγμα είναι τυχαίο.

Όταν οι στήλες σχετίζονται

Στο κεφάλαιο 32 το `city = 'Χανιά' AND region = 'Κρήτη'` υποεκτιμήθηκε οκτώ φορές. Ο planner θεωρεί από προεπιλογή ότι οι συνθήκες σε διαφορετικές στήλες είναι ανεξάρτητες και πολλαπλασιάζει τις πιθανότητες. Τα Χανιά είναι 4% των πελατών και η Κρήτη 12%, οπότε η εκτίμηση είναι 0,5%. Στην πραγματικότητα κάθε πελάτης των Χανίων είναι στην Κρήτη και το σωστό είναι 4%.

```
SQL

EXPLAIN SELECT * FROM lab.customers WHERE city = 'Χανιά' AND region = 'Κρήτη';
```

```
ΑΠΟΤΕΛΕΣΜΑ

Seq Scan on customers  (cost=0.00..6249.00 rows=924 width=98)
  Filter: ((city = 'Χανιά'::text) AND (region = 'Κρήτη'::text))
```

Το `CREATE STATISTICS` λέει στο `ANALYZE` να μαζέψει statistics για συνδυασμούς στηλών. Το είδος `dependencies` μετρά πόσο μια στήλη καθορίζει την άλλη. Το `mcv` κρατά τους πιο συχνούς συνδυασμούς. Το `ndistinct` μετρά πόσοι διαφορετικοί συνδυασμοί υπάρχουν.

```
SQL

CREATE STATISTICS lab.customers_city_region (dependencies, ndistinct, mcv)
  ON city, region FROM lab.customers;
ANALYZE lab.customers;

EXPLAIN SELECT * FROM lab.customers WHERE city = 'Χανιά' AND region = 'Κρήτη';
EXPLAIN SELECT * FROM lab.customers WHERE city = 'Αθήνα' AND region = 'Κρήτη';
```

```
ΑΠΟΤΕΛΕΣΜΑ

CREATE STATISTICS

ANALYZE

Seq Scan on customers  (cost=0.00..6249.00 rows=8047 width=98)
  Filter: ((city = 'Χανιά'::text) AND (region = 'Κρήτη'::text))

Seq Scan on customers  (cost=0.00..6249.00 rows=1 width=98)
  Filter: ((city = 'Αθήνα'::text) AND (region = 'Κρήτη'::text))
```

Τώρα η εκτίμηση για τα Χανιά είναι σωστή και για τον αδύνατο συνδυασμό Αθήνας και Κρήτης είναι μία γραμμή, η μικρότερη εκτίμηση που δίνει ποτέ ο planner. Τα extended statistics δεν δημιουργούνται αυτόματα. Τα φτιάχνεις όταν βρεις μια εκτίμηση που αστοχεί λόγω σχετιζόμενων στηλών.

Εκτιμήσεις ομάδων

Το ίδιο πρόβλημα εμφανίζεται στο `GROUP BY`. Για να εκτιμήσει πόσες ομάδες θα βγάλει ένα `GROUP BY city, region`, ο planner πολλαπλασιάζει τις διαφορετικές τιμές των στηλών, έντεκα πόλεις επί εννέα περιφέρειες. Με το `ndistinct` των extended statistics ξέρει ότι οι συνδυασμοί είναι έντεκα.

```
SQL

EXPLAIN SELECT city, region, count(*) FROM lab.customers GROUP BY city, region;
```

```
ΑΠΟΤΕΛΕΣΜΑ

HashAggregate  (cost=6749.00..6749.11 rows=11 width=41)
  Group Key: city, region
  ->  Seq Scan on customers  (cost=0.00..5249.00 rows=200000 width=33)
```

Μια υπερεκτίμηση ομάδων μπορεί να κάνει τον planner να διαλέξει ταξινόμηση αντί για `HashAggregate`, επειδή φοβάται ότι ο πίνακας κατακερματισμού δεν θα χωρέσει στη μνήμη.

Παλιά statistics

Τα statistics είναι φωτογραφία της στιγμής του `ANALYZE`. Αν ένας πίνακας αλλάξει πολύ πριν τρέξει ξανά, ο planner αποφασίζει με λάθος εικόνα. Το πιο συνηθισμένο παράδειγμα είναι ένας πίνακας που μόλις γέμισε.

```
SQL

CREATE TABLE lab.orders_2027 (LIKE lab.orders);
ANALYZE lab.orders_2027;
INSERT INTO lab.orders_2027 SELECT * FROM lab.orders WHERE id <= 200000;

EXPLAIN SELECT * FROM lab.orders_2027 WHERE status = 'cancelled';
ANALYZE lab.orders_2027;
EXPLAIN SELECT * FROM lab.orders_2027 WHERE status = 'cancelled';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

ANALYZE

INSERT 0 200000

Seq Scan on orders_2027  (cost=0.00..3026.24 rows=533 width=100)
  Filter: (status = 'cancelled'::text)

ANALYZE

Seq Scan on orders_2027  (cost=0.00..4193.00 rows=14753 width=74)
  Filter: (status = 'cancelled'::text)
```

Πριν από το δεύτερο `ANALYZE`, ο planner ήξερε μόνο ότι ο πίνακας ήταν άδειος. Εκτίμησε από το φυσικό μέγεθος του αρχείου πόσες γραμμές υπάρχουν, αλλά δεν είχε ιδέα για τις τιμές του `status` και έβαλε μια γενική εκτίμηση για το φίλτρο. Μετά το `ANALYZE` η εκτίμηση πλησίασε την πραγματικότητα. Σε διαδικασίες που φορτώνουν πολλά δεδομένα και αμέσως τα ρωτούν, ένα `ANALYZE` μετά τη φόρτωση είναι μέρος της διαδικασίας.

Ακρίβεια του δείγματος

Το μέγεθος του δείγματος και των λιστών ορίζεται από το `default_statistics_target`, από προεπιλογή 100. Κρατιούνται έως 100 συχνές τιμές και 100 κάδοι ιστογράμματος, από δείγμα 300 επί 100 γραμμές. Για μια στήλη με μεγάλη ανισορροπία μπορείς να το αυξήσεις μόνο για αυτήν.

SQL

```
ALTER TABLE lab.orders ALTER COLUMN customer_id SET STATISTICS 1000;
ANALYZE lab.orders;

SELECT attname, array_length(most_common_vals::text::text[], 1) AS mcv_entries
FROM pg_stats
WHERE schemaname = 'lab' AND tablename = 'orders' AND attname = 'customer_id';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

ANALYZE

   attname   | mcv_entries
-------------+-------------
 customer_id |        1000
(1 row)
```

Περισσότερες τιμές στη λίστα σημαίνουν ακριβέστερες εκτιμήσεις για περισσότερους πελάτες. Το κόστος είναι ένα πιο αργό `ANALYZE` και λίγο πιο αργό planning.

ΚΡΑΤΑ ΑΥΤΟ

Ο planner εκτιμά από το `pg_stats`. Ποσοστό `NULL`, διαφορετικές τιμές, λίστα συχνών τιμών, ιστόγραμμα και correlation. Θεωρεί τις στήλες ανεξάρτητες, εκτός αν φτιάξεις `CREATE STATISTICS`. Μετά από μεγάλες αλλαγές τρέχεις `ANALYZE`. Για στήλες με μεγάλη ανισορροπία αυξάνεις το `SET STATISTICS`.

Ασκήσεις

ΑΣΚΗΣΗ 34.1 Η κατανομή των καταστάσεων

Εμφάνισε από το `pg_stats` τις συχνές τιμές του `status` του `lab.orders` με τις συχνότητές τους. Σύγκρινέ τες με τα πραγματικά ποσοστά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   value   | sample_freq | real_freq
-----------+-------------+-----------
 delivered |      0.8983 |    0.8983
 cancelled |      0.0699 |    0.0697
 refunded  |      0.0299 |    0.0299
 paid      |      0.0007 |    0.0007
 pending   |      0.0007 |    0.0007
 shipped   |      0.0007 |    0.0007
(6 rows)
```

ΑΣΚΗΣΗ 34.2 Εκτίμηση για ένα διάστημα

Με `EXPLAIN ANALYZE`, σύγκρινε εκτίμηση και πραγματικότητα για τις παραγγελίες με `total > 1400`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on orders  (cost=0.00..41924.00 rows=48505 width=8) (actual time=0.090..113.086 rows=48284.00 loops=1)
  Filter: (total > '1400'::numeric)
  Rows Removed by Filter: 1951716
  Buffers: shared hit=59 read=16865 written=35
Planning:
  Buffers: shared hit=8
Planning Time: 0.039 ms
Execution Time: 114.005 ms
```

ΑΣΚΗΣΗ 34.3 Statistics για μια έκφραση

Δείξε με `EXPLAIN` την εκτίμηση για τους πελάτες του `lab.customers` με `lower(email) LIKE 'user1%'`. Φτιάξε extended statistics πάνω στην έκφραση `lower(email)`, τρέξε `ANALYZE` και δείξε ξανά την εκτίμηση. Μέτρησε και τις πραγματικές γραμμές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on customers  (cost=0.00..6249.00 rows=1000 width=4)
  Filter: (lower(email) ~~ 'user1%'::text)

CREATE STATISTICS

ANALYZE

Seq Scan on customers  (cost=0.00..6249.00 rows=111111 width=4)
  Filter: (lower(email) ~~ 'user1%'::text)

 count
--------
 111111
(1 row)
```

ΑΣΚΗΣΗ 34.4 Δύο συνθήκες που σχετίζονται

Στον `lab.orders`, οι παραγγελίες `pending` είναι όλες από τις τελευταίες ημέρες. Σύγκρινε με `EXPLAIN ANALYZE` εκτίμηση και πραγματικότητα για το `status = 'pending' AND created_at >= '2026-06-25'`. Εξήγησε την απόκλιση.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Seq Scan on orders  (cost=0.00..46924.00 rows=4 width=8) (actual time=78.475..78.705 rows=1349.00 loops=1)
  Filter: ((created_at >= '2026-06-25 00:00:00+03'::timestamp with time zone) AND (status = 'pending'::text))
  Rows Removed by Filter: 1998651
  Buffers: shared hit=123 read=16801 written=22
Planning:
  Buffers: shared hit=6 read=1
Planning Time: 0.045 ms
Execution Time: 78.737 ms
```

ΑΣΚΗΣΗ 34.5 Φόρτωση και ANALYZE

Φτιάξε κενό αντίγραφο του `lab.customers`, τρέξε `ANALYZE` και φόρτωσε τους πελάτες της Αθήνας. Δείξε με `EXPLAIN` την εκτίμηση για το `WHERE city = 'Θεσσαλονίκη'` πριν και μετά από νέο `ANALYZE`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

ANALYZE

INSERT 0 80043

Seq Scan on customers_copy  (cost=0.00..1812.00 rows=242 width=172)
  Filter: (city = 'Θεσσαλονίκη'::text)

ANALYZE

Seq Scan on customers_copy  (cost=0.00..2208.54 rows=1 width=89)
  Filter: (city = 'Θεσσαλονίκη'::text)
```

ΚΕΦΑΛΑΙΟ 35

GIN, GiST και BRIN

Το B-tree εξυπηρετεί ισότητες, διαστήματα και ταξινόμηση. Δεν βοηθά όταν η ερώτηση είναι «περιέχει», «μοιάζει» ή «επικαλύπτεται». Η PostgreSQL έχει άλλα είδη index για αυτές τις ερωτήσεις και ένα πολύ μικρό index για στήλες που ακολουθούν ήδη τη σειρά του πίνακα.

Γιατί χρειάζονται άλλα indexes

Το B-tree ταξινομεί τιμές. Ένα έγγραφο jsonb, ένας πίνακας τιμών ή ένα κείμενο δεν ταξινομούνται με τρόπο που να βοηθά το `@>` ή το `ILIKE '%λέξη%'`. Για αυτά χρειάζεται μια δομή που να οργανώνει τα **κομμάτια** μιας τιμής, τα κλειδιά ενός jsonb, τα στοιχεία ενός array, τις λέξεις ενός κειμένου ή τα trigrams του.

Όπως στα προηγούμενα κεφάλαια, τα plans είναι χωρίς παραλληλισμό.

SQL

```
SET max_parallel_workers_per_gather = 0;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET
```

GIN

Ένα **GIN** index, generalized inverted index, είναι ευρετήριο όπως στο τέλος ενός βιβλίου. Για κάθε κομμάτι κρατά τη λίστα των γραμμών που το περιέχουν. Για ένα jsonb, τα κομμάτια είναι τα κλειδιά και οι τιμές του.

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT count(*) FROM lab.events WHERE payload @> '{"campaign": "blackfriday"}';

CREATE INDEX events_payload_idx ON lab.events USING gin (payload jsonb_path_ops);

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT count(*) FROM lab.events WHERE payload @> '{"campaign": "blackfriday"}';
```

ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate (actual rows=1.00 loops=1)
  Buffers: shared read=13659 dirtied=13659 written=2701
  ->  Seq Scan on events (actual rows=10050.00 loops=1)
        Filter: (payload @> '{"campaign": "blackfriday"}'::jsonb)
        Rows Removed by Filter: 989950
        Buffers: shared read=13659 dirtied=13659 written=2701
Planning:
  Buffers: shared hit=43 read=11 dirtied=2
Planning Time: 0.275 ms
Execution Time: 281.104 ms

CREATE INDEX

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=8 read=7171 written=4726
  ->  Bitmap Heap Scan on events (actual rows=10050.00 loops=1)
        Recheck Cond: (payload @> '{"campaign": "blackfriday"}'::jsonb)
        Heap Blocks: exact=7171
        Buffers: shared hit=8 read=7171 written=4726
        ->  Bitmap Index Scan on events_payload_idx (actual rows=10050.00 loops=1)
              Index Cond: (payload @> '{"campaign": "blackfriday"}'::jsonb)
              Index Searches: 1
              Buffers: shared hit=8
Planning:
  Buffers: shared hit=17 read=6
Planning Time: 0.279 ms
Execution Time: 56.787 ms
```

Χωρίς index η βάση διάβασε κάθε σελίδα και έλεγξε κάθε έγγραφο. Με το GIN βρήκε από την αρχή τις γραμμές που περιέχουν το ζεύγος κλειδιού και τιμής. Το `USING gin` διαλέγει το είδος του index. Το `jsonb_path_ops` είναι ένα **operator class**, ένας τρόπος να σπάσει η τιμή σε κομμάτια. Υποστηρίζει μόνο το `@>` και είναι μικρότερο και γρηγορότερο από το προεπιλεγμένο, που υποστηρίζει και τα `?` και `?|`.

Τα GIN indexes είναι γρήγορα στο διάβασμα και ακριβά στο γράψιμο, αφού μια νέα γραμμή προσθέτει στοιχεία σε πολλές λίστες. Η PostgreSQL μαζεύει τις νέες εισαγωγές σε μια λίστα αναμονής και τις ενσωματώνει αργότερα.

Trigram indexes

Στο κεφάλαιο 28 είδες ότι το `LIKE '%κομμάτι%'` δεν χρησιμοποιεί B-tree. Με την επέκταση `pg_trgm`, ένα GIN index μπορεί να κρατά τα trigrams κάθε κειμένου. Ένα `ILIKE` με οποιοδήποτε μοτίβο μετατρέπεται σε αναζήτηση των trigrams του.

SQL

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE INDEX products_name_trgm_idx ON lab.products USING gin (name gin_trgm_ops);

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT id, name FROM lab.products WHERE name ILIKE '%aero c4%';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

CREATE INDEX

Bitmap Heap Scan on products (actual rows=17.00 loops=1)
  Recheck Cond: (name ~~* '%aero c4%'::text)
  Heap Blocks: exact=16
  Buffers: shared hit=31
  ->  Bitmap Index Scan on products_name_trgm_idx (actual rows=17.00 loops=1)
        Index Cond: (name ~~* '%aero c4%'::text)
        Index Searches: 1
        Buffers: shared hit=15
Planning:
  Buffers: shared hit=38 read=4 dirtied=1 written=1
Planning Time: 0.291 ms
Execution Time: 0.146 ms
```

Το index βρήκε τις γραμμές που έχουν όλα τα trigrams του μοτίβου. Το `Recheck Cond` ελέγχει μετά ότι το μοτίβο ταιριάζει πραγματικά, αφού τα ίδια trigrams μπορούν να εμφανίζονται και σε άλλη σειρά. Το ίδιο index εξυπηρετεί τους τελεστές ομοιότητας `%` του κεφαλαίου 28.

GiST

Ένα **GiST** index, generalized search tree, είναι δέντρο όπου κάθε κόμβος κρατά μια περίληψη όλων των τιμών από κάτω του. Για γεωμετρικά σχήματα η περίληψη είναι ένα ορθογώνιο που τα περικλείει. Για διαστήματα είναι ένα μεγαλύτερο διάστημα. Για κείμενα με `gist_trgm_ops` είναι ένα σύνολο trigrams.

Το GiST ξέρει να βρίσκει τιμές κοντά σε μια δεδομένη, οπότε εξυπηρετεί `ORDER BY απόσταση LIMIT n`. Αυτό λέγεται **αναζήτηση κοντινότερων γειτόνων**.

SQL

```
CREATE INDEX products_name_gist_idx ON lab.products USING gist (name gist_trgm_ops);

EXPLAIN (COSTS OFF)
SELECT name FROM lab.products ORDER BY name <-> 'Καφετιερα Aero' LIMIT 5;

SELECT name, round((name <-> 'Καφετιερα Aero')::numeric, 2) AS distance
FROM lab.products
ORDER BY name <-> 'Καφετιερα Aero'
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Limit
  ->  Index Scan using products_name_gist_idx on products
        Order By: (name <-> 'Καφετιερα Aero'::text)

          name          | distance
------------------------+----------
 Καφετιέρα Aero A70-334 |     0.52
 Καφετιέρα Aero A09-428 |     0.52
 Καφετιέρα Aero A89-327 |     0.52
 Καφετιέρα Aero 475-435 |     0.52
 Καφετιέρα Aero AAB-114 |     0.52
(5 rows)
```

Το index διαβάστηκε με τη σειρά της απόστασης, όπως δείχνει το `Order By` στο plan. Το `LIMIT` σταμάτησε μετά από πέντε γραμμές, χωρίς να υπολογιστεί η απόσταση για όλα τα προϊόντα. Η αναζήτηση χωρίς τόνο βρήκε την `Καφετιέρα`. Το GiST είναι επίσης η βάση για τα exclusion constraints του κεφαλαίου 40.

Για τα trigrams υπάρχουν και τα δύο είδη. Το GIN είναι συνήθως γρηγορότερο για `LIKE` και `%`. Το GiST είναι απαραίτητο για ταξινόμηση κατά απόσταση.

BRIN

Ένα **BRIN** index, block range index, δεν κρατά μία εγγραφή ανά γραμμή. Για κάθε ομάδα σελίδων του πίνακα, από προεπιλογή 128, κρατά μόνο την ελάχιστη και τη μέγιστη τιμή της στήλης. Είναι χρήσιμο όταν η στήλη ακολουθεί τη φυσική σειρά του πίνακα, όπως μια ημερομηνία δημιουργίας σε έναν πίνακα που μόνο μεγαλώνει. Το `correlation` του κεφαλαίου 34 λέει αν ισχύει.

SQL

```
CREATE INDEX orders_created_brin ON lab.orders USING brin (created_at);

SELECT pg_size_pretty(pg_relation_size('lab.orders_created_brin')) AS brin_size,
       pg_size_pretty(pg_relation_size('lab.orders_pkey')) AS btree_size;

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT count(*) FROM lab.orders
WHERE created_at >= '2024-03-01' AND created_at < '2024-03-02';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

 brin_size | btree_size
-----------+------------
 24 kB     | 43 MB
(1 row)

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=5 read=128 written=81
  ->  Bitmap Heap Scan on orders (actual rows=997.00 loops=1)
        Recheck Cond: ((created_at >= '2024-03-01 00:00:00+02'::timestamp with time
zone) AND (created_at < '2024-03-02 00:00:00+02'::timestamp with time zone))
        Rows Removed by Index Recheck: 14109
        Heap Blocks: lossy=128
```

```
        Buffers: shared hit=5 read=128 written=81
        ->  Bitmap Index Scan on orders_created_brin (actual rows=1280.00 loops=1)
              Index Cond: ((created_at >= '2024-03-01 00:00:00+02'::timestamp with time
zone) AND (created_at < '2024-03-02 00:00:00+02'::timestamp with time zone))
              Index Searches: 1
              Buffers: shared hit=5
Planning:
  Buffers: shared hit=19 read=1
Planning Time: 0.126 ms
Execution Time: 1.440 ms
```

Το BRIN πιάνει λίγα kilobytes για δύο εκατομμύρια γραμμές, ενώ το B-tree του primary key πιάνει δεκάδες megabytes. Για μια ημέρα η βάση βρήκε τις ομάδες σελίδων που μπορεί να περιέχουν την ημέρα και διάβασε μόνο αυτές. Το `Rows Removed by Index Recheck` είναι οι γραμμές των ίδιων σελίδων από γειτονικές ημέρες, που απορρίφθηκαν μετά την ανάγνωση.

Αν η στήλη δεν ακολουθεί τη σειρά του πίνακα, κάθε ομάδα σελίδων έχει σχεδόν όλο το εύρος τιμών και το BRIN δεν αποκλείει τίποτα. Σε τέτοιες στήλες είναι άχρηστο.

Hash indexes

Υπάρχει και το **hash** index, που εξυπηρετεί μόνο ισότητα. Είναι λίγο μικρότερο από ένα B-tree για μεγάλες τιμές, αλλά δεν εξυπηρετεί διαστήματα, ταξινόμηση ούτε μοναδικότητα. Στην πράξη το B-tree είναι σχεδόν πάντα η καλύτερη επιλογή.

Ποιο index για ποια ερώτηση

Ερώτηση	Index
ισότητα, διάστημα, ταξινόμηση	B-tree
περιέχει σε jsonb, array, tsvector	GIN
`LIKE '%κομμάτι%'`, ομοιότητα	GIN με `gin_trgm_ops`
κοντινότεροι γείτονες, διαστήματα που επικαλύπτονται	GiST
στήλη που ακολουθεί τη σειρά του πίνακα σε πολύ μεγάλο πίνακα	BRIN

ΚΡΑΤΑ ΑΥΤΟ

Το GIN είναι ανεστραμμένο ευρετήριο για τιμές που αποτελούνται από κομμάτια και εξυπηρετεί το `@>`, το full text search και τα trigrams. Το GiST είναι δέντρο περιλήψεων για επικαλύψεις και κοντινότερους γείτονες. Το BRIN κρατά ελάχιστο και μέγιστο ανά ομάδα σελίδων και είναι μικροσκοπικό, αλλά χρήσιμο μόνο όταν η στήλη ακολουθεί τη φυσική σειρά του πίνακα.

Ασκήσεις

ΑΣΚΗΣΗ 35.1 Events από κινητό για ένα προϊόν

Με το GIN index της θεωρίας, δείξε το plan με `EXPLAIN (COSTS OFF)` και μέτρησε τα events από `mobile` για το προϊόν 777.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate
  ->  Bitmap Heap Scan on events
        Recheck Cond: (payload @> '{"device": "mobile", "product_id": 777}'::jsonb)
        ->  Bitmap Index Scan on events_payload_idx
              Index Cond: (payload @> '{"device": "mobile", "product_id": 777}'::jsonb)

 count
-------
     7
(1 row)
```

ΑΣΚΗΣΗ 35.2 Αναζήτηση μέσα στην περιγραφή

Φτιάξε trigram GIN index στην περιγραφή του `lab.products` και δείξε με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)` ότι το χρησιμοποιεί η αναζήτηση `description ILIKE '%ξύλο οξιάς%'`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=1152
  ->  Bitmap Heap Scan on products (actual rows=10108.00 loops=1)
        Recheck Cond: (description ~~* '%ξύλο οξιάς%'::text)
        Heap Blocks: exact=1116
        Buffers: shared hit=1152
        ->  Bitmap Index Scan on products_description_trgm_idx (actual rows=10108.00 loops=1)
              Index Cond: (description ~~* '%ξύλο οξιάς%'::text)
              Index Searches: 1
              Buffers: shared hit=36
Planning:
  Buffers: shared hit=30 read=5 dirtied=2
Planning Time: 0.578 ms
Execution Time: 13.367 ms
```

ΑΣΚΗΣΗ 35.3 BRIN στα events

Φτιάξε BRIN index στο `occurred_at` του `lab.events` και B-tree index στην ίδια στήλη. Σύγκρινε τα μεγέθη τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

CREATE INDEX

      relname      | size
-------------------+-------
 events_time_brin  | 24 kB
 events_time_btree | 21 MB
(2 rows)
```

ΑΣΚΗΣΗ 35.4 Full text search με index

Πρόσθεσε στον `lab.products` stored generated column `search` με `to_tsvector('greek', name || ' ' || description)` και GIN index πάνω της. Δείξε το plan και μέτρησε τα προϊόντα που ταιριάζουν στην αναζήτηση `αθόρυβο γραφείο`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

CREATE INDEX

Aggregate
  ->  Bitmap Heap Scan on products
        Recheck Cond: (search @@ '''αθορυβ'' & ''γραφει'''::tsquery)
        ->  Bitmap Index Scan on products_search_idx
              Index Cond: (search @@ '''αθορυβ'' & ''γραφει'''::tsquery)

 count
-------
   904
(1 row)
```

ΑΣΚΗΣΗ 35.5 Κοντινότερα ονόματα με λάθη

Με το GiST index της θεωρίας, βρες τα πέντε προϊόντα με όνομα πιο κοντά στο ανορθόγραφο `Ακουστηκα Nova`, με την απόστασή τους. Δείξε και το plan.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Limit
  ->  Index Scan using products_name_gist_idx on products
        Order By: (name <-> 'Ακουστηκα Nova'::text)

          name          | distance
------------------------+----------
 Ακουστικά Nova 9DF-977 |     0.58
 Ακουστικά Nova 2E0-273 |     0.58
 Ακουστικά Nova 93E-957 |     0.58
 Ακουστικά Nova 328-349 |     0.58
 Ακουστικά Nova 33D-300 |     0.58
(5 rows)
```

ΚΕΦΑΛΑΙΟ 36

Τεχνικές για γρήγορα queries

Τα περισσότερα αργά queries δεν χρειάζονται μαγικές ρυθμίσεις. Χρειάζονται ένα index που ταιριάζει, μια συνθήκη που το αφήνει να χρησιμοποιηθεί ή μια διατύπωση που δεν ζητά από τη βάση περισσότερη δουλειά από όση χρειάζεται. Αυτό το κεφάλαιο μαζεύει τα μοτίβα που θα συναντάς πιο συχνά.

Πρώτα μέτρα

Η βελτιστοποίηση ξεκινά από το ποια queries κοστίζουν. Η επέκταση `pg_stat_statements` κρατά για κάθε διαφορετικό query πόσες φορές εκτελέστηκε, πόσο χρόνο πήρε συνολικά και πόσες σελίδες διάβασε. Χρειάζεται να φορτωθεί κατά την εκκίνηση του server με τη ρύθμιση `shared_preload_libraries = 'pg_stat_statements'`. Μετά ενεργοποιείται σε κάθε βάση.

SQL

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
SELECT pg_stat_statements_reset() IS NOT NULL AS reset;

SELECT count(*) FROM lab.orders WHERE customer_id = 42;
SELECT count(*) FROM lab.orders WHERE customer_id = 4242;
SELECT count(*) FROM lab.orders WHERE status = 'pending';

SELECT calls, rows, shared_blks_hit + shared_blks_read AS pages,
       left(query, 60) AS query
FROM pg_stat_statements
WHERE dbid = (SELECT oid FROM pg_database WHERE datname = current_database())
  AND query LIKE '%lab.orders%' AND query NOT LIKE '%pg_stat%'
ORDER BY pages DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

 reset
-------
 t
(1 row)

 count
-------
   345
(1 row)

 count
-------
    41
(1 row)
```

```
 count
-------
  1349
(1 row)

 calls | rows | pages |                          query
-------+------+-------+-------------------------------------------------------
     2 |    2 | 33848 | SELECT count(*) FROM lab.orders WHERE customer_id = $1
     1 |    1 | 16924 | SELECT count(*) FROM lab.orders WHERE status = $1
(2 rows)
```

Τα δύο πρώτα queries έγιναν ένα, με `$1` στη θέση της τιμής. Το φίλτρο στο `dbid` κρατά μόνο τα queries της τρέχουσας βάσης, αφού το view δείχνει όλο τον server. Το `pg_stat_statements` ομαδοποιεί τα queries που διαφέρουν μόνο στις σταθερές, οπότε ένα query που τρέχει χιλιάδες φορές με διαφορετικές παραμέτρους εμφανίζεται μία φορά με το συνολικό του κόστος. Σε ένα πραγματικό σύστημα ταξινομείς κατά `total_exec_time` και ξεκινάς από την κορυφή. Ένα query των 5 ms που εκτελείται ένα εκατομμύριο φορές την ημέρα κοστίζει περισσότερο από ένα query των 10 δευτερολέπτων που τρέχει μία φορά.

Για τα υπόλοιπα παραδείγματα κλείνουμε ξανά τον παραλληλισμό.

SQL

```
SET max_parallel_workers_per_gather = 0;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET
```

Keyset pagination

Στο κεφάλαιο 4 είδες ότι το `OFFSET` δεν είναι δωρεάν. Για να δώσει τη σελίδα που ξεκινά από τη γραμμή 500.000, η βάση διαβάζει και πετά τις 500.000 προηγούμενες.

SQL

```
CREATE INDEX orders_created_id_idx ON lab.orders (created_at, id);

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT id, created_at, total FROM lab.orders
ORDER BY created_at, id
LIMIT 20 OFFSET 500000;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Limit (actual rows=20.00 loops=1)
  Buffers: shared hit=173401 read=6150 written=971
  ->  Index Scan using orders_created_id_idx on orders (actual rows=500020.00 loops=1)
        Index Searches: 1
        Buffers: shared hit=173401 read=6150 written=971
Planning:
  Buffers: shared hit=36 read=10 dirtied=1 written=6
Planning Time: 0.183 ms
Execution Time: 73.482 ms
```

Το **keyset pagination** θυμάται την τελευταία γραμμή της προηγούμενης σελίδας και ζητά τις γραμμές μετά από αυτήν. Η σύγκριση `(created_at, id) > (τιμή, τιμή)` είναι σύγκριση **row values**. Συγκρίνει πρώτα το `created_at` και, σε ισοβαθμία, το `id`, ακριβώς όπως είναι ταξινομημένο το index.

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT id, created_at, total FROM lab.orders
WHERE (created_at, id) > ('2022-05-18 10:00+03', 500000)
ORDER BY created_at, id
LIMIT 20;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Limit (actual rows=20.00 loops=1)
  Buffers: shared hit=8 read=4
  ->  Index Scan using orders_created_id_idx on orders (actual rows=20.00 loops=1)
        Index Cond: (ROW(created_at, id) > ROW('2022-05-18 10:00:00+03'::timestamp with time zone, 500000))
        Index Searches: 1
        Buffers: shared hit=8 read=4
Planning Time: 0.039 ms
Execution Time: 0.034 ms
```

Η βάση κατέβηκε στο index κατευθείαν στο σωστό σημείο και διάβασε 20 γραμμές. Ο χρόνος είναι ίδιος για την πρώτη και για τη χιλιοστή σελίδα. Το τίμημα είναι ότι δεν μπορείς να πηδήξεις στη σελίδα 1000 χωρίς να περάσεις από τις προηγούμενες. Για λίστες με «επόμενο» και «προηγούμενο» και για infinite scroll αυτό δεν είναι πρόβλημα.

Top N ανά ομάδα

Οι τρεις πιο πρόσφατες παραγγελίες κάθε πελάτη γράφονται με `LATERAL` ή με `row_number`, όπως είδες στα κεφάλαια 14 και 23. Με το κατάλληλο index, η διαφορά στην απόδοση είναι μεγάλη.

SQL

```
CREATE INDEX orders_customer_created_idx ON lab.orders (customer_id, created_at DESC);

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT c.id, recent.id
FROM lab.customers AS c
CROSS JOIN LATERAL (SELECT o.id FROM lab.orders AS o
                    WHERE o.customer_id = c.id
                    ORDER BY o.created_at DESC
                    LIMIT 3) AS recent
WHERE c.id <= 1000;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

Nested Loop (actual rows=3000.00 loops=1)
  Buffers: shared hit=4763 read=1238
  ->  Index Only Scan using customers_pkey on customers c (actual rows=1000.00 loops=1)
        Index Cond: (id <= 1000)
        Heap Fetches: 0
        Index Searches: 1
        Buffers: shared hit=4 read=2
  ->  Limit (actual rows=3.00 loops=1000)
        Buffers: shared hit=4759 read=1236
        ->  Index Scan using orders_customer_created_idx on orders o (actual rows=3.00 loops=1000)
              Index Cond: (customer_id = c.id)
              Index Searches: 1000
              Buffers: shared hit=4759 read=1236
Planning:
  Buffers: shared hit=92 read=10 dirtied=3
Planning Time: 0.493 ms
Execution Time: 5.629 ms
```

Για κάθε πελάτη η βάση διάβασε τρεις εγγραφές του index και σταμάτησε. Με `row_number`, διαβάζει όλες τις παραγγελίες των χιλίων πελατών, τις ταξινομεί και μετά πετά όσες δεν είναι στις τρεις πρώτες.

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT customer_id, id
FROM (SELECT customer_id, id,
             row_number() OVER (PARTITION BY customer_id ORDER BY created_at DESC) AS rn
      FROM lab.orders
      WHERE customer_id <= 1000) AS t
WHERE rn <= 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Subquery Scan on t (actual rows=3000.00 loops=1)
  Buffers: shared hit=5296 read=12172 written=477, temp read=593 written=595
  ->  WindowAgg (actual rows=3000.00 loops=1)
        Window: w1 AS (PARTITION BY orders.customer_id ORDER BY orders.created_at ROWS UNBOUNDED PRECEDING)
        Run Condition: (row_number() OVER w1 <= 3)
        Storage: Memory  Maximum Storage: 17kB
        Buffers: shared hit=5296 read=12172 written=477, temp read=593 written=595
        ->  Sort (actual rows=142406.00 loops=1)
              Sort Key: orders.customer_id, orders.created_at DESC
              Sort Method: external merge  Disk: 4744kB
              Buffers: shared hit=5296 read=12172 written=477, temp read=593 written=595
              ->  Bitmap Heap Scan on orders (actual rows=142406.00 loops=1)
                    Recheck Cond: (customer_id <= 1000)
                    Heap Blocks: exact=16920
```

```
                    Buffers: shared hit=5296 read=12172 written=477
                    ->  Bitmap Index Scan on orders_customer_created_idx (actual rows=142406.00 loops=1)
                          Index Cond: (customer_id <= 1000)
                          Index Searches: 1
                          Buffers: shared hit=473 read=75
Planning:
  Buffers: shared hit=4
Planning Time: 0.057 ms
Execution Time: 70.153 ms
```

Οι πελάτες με μικρό `id` έχουν δεκάδες ή εκατοντάδες παραγγελίες ο καθένας, οπότε το `row_number` διάβασε και ταξινόμησε πάνω από εκατό χιλιάδες γραμμές για να κρατήσει τρεις χιλιάδες. Όταν οι ομάδες είναι μεγάλες και θέλεις λίγες γραμμές από καθεμία, το `LATERAL` με index κερδίζει. Όταν θέλεις σχεδόν όλες τις γραμμές, το `row_number` είναι απλούστερο και εξίσου γρήγορο.

Ύπαρξη ή πλήθος

Ένα συχνό λάθος είναι να μετράς για να μάθεις αν υπάρχει κάτι. Το `count(*) > 0` πρέπει να βρει όλες τις γραμμές. Το `EXISTS` σταματά στην πρώτη.

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT count(*) > 0 AS has_orders FROM lab.orders WHERE customer_id = 1;

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT EXISTS (SELECT 1 FROM lab.orders WHERE customer_id = 1) AS has_orders;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=4 read=17
  ->  Index Only Scan using orders_customer_created_idx on orders (actual rows=4460.00 loops=1)
        Index Cond: (customer_id = 1)
        Heap Fetches: 0
        Index Searches: 1
        Buffers: shared hit=4 read=17
Planning Time: 0.043 ms
Execution Time: 0.276 ms

Result (actual rows=1.00 loops=1)
  Buffers: shared hit=4
  InitPlan 1
    ->  Index Only Scan using orders_customer_created_idx on orders (actual rows=1.00 loops=1)
          Index Cond: (customer_id = 1)
          Heap Fetches: 0
          Index Searches: 1
          Buffers: shared hit=4
Planning Time: 0.017 ms
Execution Time: 0.009 ms
```

Ο πελάτης 1 έχει χιλιάδες παραγγελίες. Το `count` διάβασε όλες τις εγγραφές του στο index, το `EXISTS` μία.

Πολλά queries αντί για ένα

Ένας κώδικας εφαρμογής που φέρνει μια λίστα και μετά κάνει ένα query για κάθε στοιχείο της στέλνει στη βάση χιλιάδες μικρά queries. Αυτό λέγεται πρόβλημα **N+1**. Κάθε query είναι γρήγορο, αλλά το σύνολο των round trips ανάμεσα σε εφαρμογή και βάση είναι αργό. Η λύση είναι ένα query για όλη τη λίστα, με `= ANY` και array παραμέτρων ή με join.

```sql
SELECT customer_id, count(*) AS orders
FROM lab.orders
WHERE customer_id = ANY (ARRAY[150000, 150001, 150002, 150003])
GROUP BY customer_id
ORDER BY customer_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 customer_id | orders
-------------+--------
      150000 |     10
      150001 |      2
      150002 |      5
      150003 |      9
(4 rows)
```

Η εφαρμογή στέλνει τη λίστα ως μία παράμετρο τύπου array. Το query μένει ίδιο όποιο κι αν είναι το μήκος της λίστας, οπότε και το `pg_stat_statements` το βλέπει ως ένα.

Ακριβές ή περίπου

Το `count(*)` σε έναν μεγάλο πίνακα πρέπει να διαβάσει όλες τις γραμμές ή όλο ένα index, αφού στην PostgreSQL κάθε transaction μπορεί να βλέπει διαφορετικό πλήθος. Για ένα «περίπου 2 εκατομμύρια αποτελέσματα» σε μια σελίδα αρκεί η εκτίμηση του καταλόγου, που ενημερώνεται από το `VACUUM` και το `ANALYZE`.

```sql
SELECT reltuples::bigint AS estimated FROM pg_class WHERE oid = 'lab.orders'::regclass;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 estimated
-----------
   2000000
(1 row)
```

Γενικοί κανόνες

Σύμπτωμα	Συνήθης αιτία	Διόρθωση
`Seq Scan` με μεγάλο `Rows Removed by Filter`	λείπει index ή η συνθήκη δεν είναι sargable	index, στήλη μόνη της στη σύγκριση
μεγάλη απόκλιση `rows` εκτίμησης και πραγματικότητας	statistics	`ANALYZE`, `CREATE STATISTICS`, `SET STATISTICS`
`Sort Method: external merge`	λίγη μνήμη	`work_mem` για το query, index που δίνει τη σειρά

Σύμπτωμα	Συνήθης αιτία	Διόρθωση
`Heap Fetches` σε `Index Only Scan`	visibility map	`VACUUM`, ρύθμιση autovacuum
αργό `OFFSET`	η βάση διαβάζει όλες τις προηγούμενες γραμμές	keyset pagination
χιλιάδες όμοια queries	N+1	ένα query με `ANY` ή join

ΚΡΑΤΑ ΑΥΤΟ

Μέτρα πρώτα με `pg_stat_statements` και `EXPLAIN ANALYZE`. Σελιδοποίησε με keyset. Για λίγες γραμμές από πολλές ομάδες χρησιμοποίησε `LATERAL` με index. Ρώτα ύπαρξη με `EXISTS`, όχι με `count`. Στείλε λίστες ως array σε ένα query. Για περίπου πλήθη διάβασε τον κατάλογο.

Ασκήσεις

Στις ασκήσεις ισχύει το `max_parallel_workers_per_gather = 0` και υπάρχουν τα indexes της θεωρίας.

ΑΣΚΗΣΗ 36.1 Η επόμενη σελίδα

Η πρώτη σελίδα των παραγγελιών του πελάτη 777, από την πιο πρόσφατη, με πέντε παραγγελίες ανά σελίδα, τελειώνει στην παραγγελία με `created_at` και `id` που θα βρεις με το πρώτο query. Γράψε το query της δεύτερης σελίδας με keyset pagination.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   id    |          created_at
---------+-------------------------------
 1969541 | 2026-05-31 11:57:26.260093+03
 1936268 | 2026-04-28 01:18:00.577505+03
 1917610 | 2026-04-09 09:31:01.185311+03
 1910564 | 2026-04-02 07:31:53.880285+03
 1892564 | 2026-03-15 05:36:54.261034+02
(5 rows)

   id    |          created_at
---------+-------------------------------
 1871039 | 2026-02-21 14:55:00.881456+02
 1855762 | 2026-02-06 06:20:40.011039+02
 1840112 | 2026-01-21 14:41:56.191803+02
 1810734 | 2025-12-23 03:20:47.100618+02
 1782802 | 2025-11-25 02:05:12.63263+02
(5 rows)
```

ΑΣΚΗΣΗ 36.2 Σύγκριση σελιδοποίησης

Με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)`, σύγκρινε το query που φέρνει τις παραγγελίες 1.000.001 έως 1.000.010 κατά `id` με `OFFSET` και με keyset στο `id`. Εντόπισε τη διαφορά στα buffers.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Limit (actual rows=10.00 loops=1)
  Buffers: shared hit=4336 read=6964 written=64
  ->  Index Scan using orders_pkey on orders (actual rows=1000010.00 loops=1)
        Index Searches: 1
        Buffers: shared hit=4336 read=6964 written=64
Planning Time: 0.011 ms
Execution Time: 79.683 ms

Limit (actual rows=10.00 loops=1)
  Buffers: shared hit=3 read=1
  ->  Index Scan using orders_pkey on orders (actual rows=10.00 loops=1)
        Index Cond: (id > 1000000)
        Index Searches: 1
        Buffers: shared hit=3 read=1
Planning Time: 0.039 ms
Execution Time: 0.021 ms
```

ΑΣΚΗΣΗ 36.3 Υπάρχει ακριβή παραγγελία

Γράψε με `EXISTS` query που επιστρέφει για τους πελάτες 1 έως 5 αν έχουν τουλάχιστον μία παραγγελία πάνω από 1490 ευρώ. Δείξε το plan με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Index Only Scan using customers_pkey on customers c (actual rows=5.00 loops=1)
  Index Cond: (id <= 5)
  Heap Fetches: 0
  Index Searches: 1
  Buffers: shared hit=913 read=149 written=7
  SubPlan 1
    ->  Bitmap Heap Scan on orders o (actual rows=1.00 loops=5)
          Recheck Cond: (customer_id = c.id)
          Filter: (total > '1490'::numeric)
          Rows Removed by Filter: 206
          Heap Blocks: exact=983
          Buffers: shared hit=909 read=149 written=7
          ->  Bitmap Index Scan on orders_customer_created_idx (actual rows=2012.20 loops=5)
                Index Cond: (customer_id = c.id)
                Index Searches: 5
                Buffers: shared hit=31 read=22
Planning:
  Buffers: shared hit=5 read=6
Planning Time: 0.099 ms
Execution Time: 2.089 ms
```

ΑΣΚΗΣΗ 36.4 Μια λίστα αντί για πολλά queries

Μια σελίδα δείχνει τους πελάτες 150010 έως 150014 και για τον καθένα το ποσό της πιο πρόσφατης παραγγελίας. Γράψε ένα query που δίνει και τους πέντε μαζί, με `= ANY` και `LATERAL`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
   id    | first_name |  total  | created_at
---------+------------+---------+------------
 150010 | Φωτεινή    |  372.28 | 2024-11-25
 150011 | Στέλιος    |  378.38 | 2026-04-04
 150012 | Νίκος      |  171.03 | 2025-12-27
 150013 | Αλέξανδρος |   14.13 | 2025-10-29
 150014 | Ζωή        | 1281.25 | 2024-11-25
(5 rows)
```

ΑΣΚΗΣΗ 36.5 Εκτίμηση ή ακρίβεια

Σύγκρινε την εκτίμηση του καταλόγου για το πλήθος των γραμμών του `lab.events` με το ακριβές `count(*)`. Δείξε πόσες σελίδες διάβασε το `count(*)`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 estimated
-----------
   1000000
(1 row)

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared read=13659 dirtied=13659 written=2554
  ->  Seq Scan on events (actual rows=1000000.00 loops=1)
        Buffers: shared read=13659 dirtied=13659 written=2554
Planning:
  Buffers: shared hit=16 read=3 dirtied=1
Planning Time: 0.128 ms
Execution Time: 60.792 ms
```

ΜΕΡΟΣ VI

Concurrency και προχωρημένα εργαλεία

Πολλοί χρήστες γράφουν ταυτόχρονα. Εδώ βλέπουμε τι βλέπει κάθε transaction, πότε περιμένει και πότε αποτυγχάνει. Στο τέλος, λογική μέσα στη βάση, διαστήματα χωρίς επικαλύψεις, partitioning και δικαιώματα ανά γραμμή.

ΚΕΦΑΛΑΙΟ 37

MVCC και isolation levels

Κάθε transaction βλέπει ένα **snapshot**, μια φωτογραφία της βάσης που αποφασίζει ποιες εκδόσεις γραμμών είναι ορατές. Το isolation level ορίζει πότε παίρνεται η φωτογραφία και τι γίνεται όταν δύο transactions αλλάζουν τα ίδια δεδομένα. Οι αναγνώστες δεν περιμένουν ποτέ τους γράφοντες.

Snapshots

Στο κεφάλαιο 29 είδες ότι κάθε αλλαγή γράφει νέα έκδοση γραμμής με το `xmin` του transaction που την έγραψε. Ένα snapshot είναι η λίστα των transactions που είχαν ολοκληρωθεί τη στιγμή που πάρθηκε. Μια έκδοση είναι ορατή αν το transaction που τη δημιούργησε είχε κάνει commit πριν από το snapshot και κανένα transaction που είχε κάνει commit πριν από το snapshot δεν τη διέγραψε.

Αυτός είναι ο λόγος που στην PostgreSQL ένα `SELECT` δεν περιμένει ποτέ ένα `UPDATE`. Ο αναγνώστης βλέπει την παλιά έκδοση, που είναι ακόμη στη σελίδα. Ο γράφων φτιάχνει τη νέα χωρίς να ενοχλεί κανέναν.

READ COMMITTED

Το προεπιλεγμένο isolation level είναι το **READ COMMITTED**. Κάθε εντολή μέσα στο transaction παίρνει νέο snapshot. Βλέπει ό,τι είχε κάνει commit όταν ξεκίνησε η ίδια η εντολή.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
SELECT stock FROM products WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

 stock
-------
    42
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE products SET stock = 40 WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
SELECT stock FROM products WHERE id = 1;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
 stock
-------
    40
(1 row)

COMMIT
```

Η συνεδρία Α διάβασε δύο διαφορετικές τιμές μέσα στο ίδιο transaction. Αυτό λέγεται **non repeatable read**. Για τις περισσότερες εφαρμογές είναι αποδεκτό, αφού κάθε εντολή βλέπει μια συνεπή εικόνα και η πιο πρόσφατη πληροφορία είναι συνήθως αυτή που θέλεις.

Σχήμα 37.1 · Στο READ COMMITTED κάθε εντολή βλέπει τα πιο πρόσφατα commits. Στο REPEATABLE READ όλο το transaction βλέπει το snapshot της πρώτης του εντολής.

REPEATABLE READ

Στο **REPEATABLE READ** το snapshot παίρνεται στην πρώτη εντολή του transaction και μένει ίδιο ως το τέλος. Ό,τι κι αν κάνουν οι άλλοι, το transaction βλέπει την ίδια εικόνα.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT stock FROM products WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

 stock
-------
    40
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE products SET stock = 38 WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
SELECT stock FROM products WHERE id = 1;
UPDATE products SET stock = stock - 1 WHERE id = 1;
ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
 stock
-------
    40
(1 row)

ERROR:  could not serialize access due to concurrent update

ROLLBACK
```

Το δεύτερο SELECT έδωσε την ίδια τιμή με το πρώτο. Όταν όμως η συνεδρία Α προσπάθησε να αλλάξει τη γραμμή, απέτυχε. Η γραμμή είχε αλλάξει από άλλο transaction μετά το snapshot της Α. Αν η Α έγραφε με βάση το 40 που βλέπει, θα έσβηνε την αλλαγή της Β. Στο REPEATABLE READ η PostgreSQL προτιμά να αποτύχει το transaction. Η εφαρμογή πρέπει να το ξαναρχίσει από την αρχή.

Το REPEATABLE READ είναι κατάλληλο για αναφορές που τρέχουν πολλά queries και πρέπει να βλέπουν συνεπή εικόνα, για παράδειγμα σύνολα που πρέπει να συμφωνούν μεταξύ τους.

Χαμένες ενημερώσεις

Το πιο συνηθισμένο λάθος concurrency δεν είναι θέμα isolation level. Είναι κώδικας εφαρμογής που διαβάζει μια τιμή, υπολογίζει τη νέα και τη γράφει. Δύο πελάτες αγοράζουν ταυτόχρονα το προϊόν 2.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
SELECT stock FROM products WHERE id = 2;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

 stock
-------
    18
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
BEGIN;
SELECT stock FROM products WHERE id = 2;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
BEGIN

 stock
-------
    18
(1 row)
```

Και οι δύο βλέπουν 18 και υπολογίζουν ότι το νέο απόθεμα είναι 17.

SQL · ΣΥΝΕΔΡΙΑ Α

```
UPDATE products SET stock = 17 WHERE id = 2;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE products SET stock = 17 WHERE id = 2;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
-- η εντολή περιμένει
```

Η συνεδρία Β περιμένει. Η Α έχει αλλάξει τη γραμμή χωρίς να κάνει commit και κρατά ένα **row lock** πάνω της. Δύο transactions δεν μπορούν να αλλάζουν ταυτόχρονα την ίδια γραμμή. Όταν η Α κάνει commit, η Β συνεχίζει.

SQL · ΣΥΝΕΔΡΙΑ Α

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
COMMIT
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
COMMIT;
SELECT stock FROM products WHERE id = 2;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
COMMIT

 stock
-------
    17
(1 row)
```

Πουλήθηκαν δύο τεμάχια και το απόθεμα μειώθηκε κατά ένα. Η ενημέρωση της Α χάθηκε. Αυτό λέγεται **lost update** και κανένα από τα δύο transactions δεν έκανε κάτι λάθος από την πλευρά της βάσης. Το λάθος ήταν ότι η τιμή υπολογίστηκε έξω από τη βάση.

Η διόρθωση είναι να αφήσεις τη βάση να υπολογίσει τη νέα τιμή. Στο READ COMMITTED, όταν ένα `UPDATE` περιμένει μια γραμμή και αυτή αλλάξει, η PostgreSQL ξαναδιαβάζει τη νέα έκδοση και ξαναυπολογίζει την έκφραση πάνω της.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
UPDATE products SET stock = stock - 1 WHERE id = 2;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE products SET stock = stock - 1 WHERE id = 2 RETURNING stock;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
-- η εντολή περιμένει
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
COMMIT
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
 stock
-------
    15
(1 row)

UPDATE 1
```

Η Β περίμενε, διάβασε το 16 που έγραψε η Α και έγραψε 15. Τίποτα δεν χάθηκε. Όταν ο υπολογισμός είναι πιο σύνθετος από μια αφαίρεση, κλειδώνεις τη γραμμή πριν τη διαβάσεις, με `SELECT ... FOR UPDATE`, που είναι το θέμα του επόμενου κεφαλαίου.

SERIALIZABLE και write skew

Υπάρχει μια πιο λεπτή ανωμαλία, που ούτε το REPEATABLE READ δεν πιάνει. Ο κανόνας του καταστήματος λέει ότι η κατηγορία Ήχος πρέπει να έχει τουλάχιστον τρία ενεργά προϊόντα. Δύο διαχειριστές απενεργοποιούν ταυτόχρονα από ένα διαφορετικό προϊόν, αφού πρώτα ελέγξουν τον κανόνα.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT count(*) FROM products WHERE category_id = 10 AND active;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

 count
-------
     4
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT count(*) FROM products WHERE category_id = 10 AND active;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
BEGIN

 count
-------
     4
(1 row)
```

Και οι δύο βλέπουν τέσσερα και κρίνουν ότι μπορούν να αφαιρέσουν ένα.

SQL · ΣΥΝΕΔΡΙΑ Α

```
UPDATE products SET active = false WHERE id = 20;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE products SET active = false WHERE id = 21;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1
```

Οι δύο αλλάζουν διαφορετικές γραμμές, οπότε κανείς δεν περιμένει κανέναν.

SQL · ΣΥΝΕΔΡΙΑ Α

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
COMMIT
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

Αν περνούσαν και τα δύο, η κατηγορία θα έμενε με δύο ενεργά προϊόντα και ο κανόνας θα παραβιαζόταν, παρόλο που κάθε transaction τον έλεγξε. Αυτό είναι το **write skew**. Στο REPEATABLE READ θα περνούσαν και τα δύο. Το **SERIALIZABLE** παρακολουθεί ποια transaction

διάβασε δεδομένα που άλλαξε ένα άλλο. Όταν βρει κύκλο εξαρτήσεων που δεν θα μπορούσε να συμβεί αν τα transactions εκτελούνταν το ένα μετά το άλλο, αναγκάζει ένα από αυτά να αποτύχει.

> **ΠΑΓΙΔΑ**
> Στο REPEATABLE READ και στο SERIALIZABLE, η εφαρμογή πρέπει να είναι έτοιμη να ξαναεκτελέσει ολόκληρο το transaction όταν πάρει σφάλμα serialization, με κωδικό `40001`. Ένα SERIALIZABLE χωρίς λογική επανάληψης είναι απλώς ένας τρόπος να αποτυγχάνουν αιτήματα.

Ποιο isolation level

Επίπεδο	Snapshot	Τι προλαβαίνει
READ COMMITTED	νέο σε κάθε εντολή	τα dirty reads
REPEATABLE READ	ένα για όλο το transaction	και τα non repeatable reads και τα lost updates στις ίδιες γραμμές
SERIALIZABLE	ένα, με έλεγχο εξαρτήσεων	όλες τις ανωμαλίες, μαζί με το write skew

Η PostgreSQL δέχεται και το `READ UNCOMMITTED` του προτύπου, αλλά το εκτελεί ως READ COMMITTED. Ένα transaction δεν βλέπει ποτέ αλλαγές που δεν έχουν γίνει commit.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Κάθε transaction βλέπει ένα snapshot και οι αναγνώστες δεν περιμένουν τους γράφοντες. Στο READ COMMITTED το snapshot ανανεώνεται σε κάθε εντολή. Στο REPEATABLE READ μένει σταθερό και οι ταυτόχρονες αλλαγές στην ίδια γραμμή αποτυγχάνουν. Το SERIALIZABLE πιάνει και το write skew. Υπολόγιζε τις νέες τιμές μέσα στη βάση, όχι στην εφαρμογή.

Ασκήσεις

ΑΣΚΗΣΗ 37.1 Το isolation level ενός transaction

Μέσα σε transaction με `REPEATABLE READ`, εμφάνισε την τρέχουσα τιμή της ρύθμισης `transaction_isolation`. Μετά εμφάνισέ την ξανά εκτός transaction.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

 transaction_isolation
-----------------------
 repeatable read
(1 row)

COMMIT

 transaction_isolation
-----------------------
 read committed
(1 row)
```

ΑΣΚΗΣΗ 37.2 Ένα snapshot για όλο το transaction

Σε transaction με `REPEATABLE READ`, εμφάνισε δύο φορές το `pg_current_snapshot()` με μια αλλαγή σε κάποια γραμμή ανάμεσα. Σύγκρινε τις δύο τιμές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

 has_snapshot
--------------
 t
(1 row)

SELECT 1

UPDATE 1

 same_snapshot
---------------
 t
(1 row)

ROLLBACK
```

ΑΣΚΗΣΗ 37.3 Αγορά χωρίς υπερπώληση

Γράψε ένα `UPDATE` που μειώνει το απόθεμα του προϊόντος 26 κατά 2 μόνο αν υπάρχουν αρκετά τεμάχια. Να επιστρέφει το νέο απόθεμα. Δοκίμασέ το δύο φορές στο ίδιο transaction.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 stock
-------
     1
(1 row)

UPDATE 1

 stock
-------
(0 rows)

UPDATE 0
```

ΑΣΚΗΣΗ 37.4 Ποιο transaction έγραψε τη γραμμή

Σε transaction, αύξησε κατά 1 το απόθεμα του προϊόντος 4. Εμφάνισε το `xmin` της γραμμής ως κείμενο και τον αριθμό του τρέχοντος transaction με `pg_current_xact_id()`. Έλεγξε αν είναι ίσα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
UPDATE 1

 written_by_me
---------------
 t
(1 row)
```

ΑΣΚΗΣΗ 37.5 Σταθερός χρόνος μέσα στο transaction

Μέσα σε transaction, σύγκρινε το `now()` με το `transaction_timestamp()` και το `statement_timestamp()` με το `clock_timestamp()`, σε δύο διαδοχικές εντολές με μια μικρή καθυστέρηση ανάμεσα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 now_is_transaction_start
--------------------------
 t
(1 row)

 pg_sleep
----------

(1 row)

 statement_is_later | clock_is_latest
--------------------+-----------------
 t                  | t
(1 row)
```

ΚΕΦΑΛΑΙΟ 38

Locks, SKIP LOCKED και deadlocks

Το MVCC αφήνει τους αναγνώστες να μην περιμένουν. Οι γράφοντες όμως που αλλάζουν την ίδια γραμμή πρέπει να μπουν σε σειρά. Αυτό γίνεται με locks. Όταν ξέρεις ποια locks παίρνει κάθε εντολή, μπορείς να φτιάξεις ουρές εργασιών, να αποφύγεις deadlocks και να αλλάζεις σχήμα χωρίς να σταματάς την εφαρμογή.

Row locks

Κάθε `UPDATE` και `DELETE` κλειδώνει τις γραμμές που αλλάζει μέχρι το τέλος του transaction. Το είδες στο προηγούμενο κεφάλαιο, όπου η δεύτερη συνεδρία περίμενε. Μπορείς να κλειδώσεις γραμμές και όταν τις διαβάζεις, με `SELECT ... FOR UPDATE`. Είναι ο τρόπος να διαβάσεις μια τιμή, να αποφασίσεις στην εφαρμογή και να γράψεις, χωρίς να αλλάξει κανείς τη γραμμή στο μεταξύ.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
SELECT id, stock FROM products WHERE id = 5 FOR UPDATE;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

 id | stock
----+-------
  5 |    25
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
BEGIN;
SELECT id, stock FROM products WHERE id = 5 FOR UPDATE;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
-- η εντολή περιμένει
```

Η συνεδρία Β περιμένει στο ίδιο `SELECT`. Η Α αποφασίζει ότι το απόθεμα αρκεί, το μειώνει και κάνει commit.

SQL · ΣΥΝΕΔΡΙΑ Α

```
UPDATE products SET stock = stock - 3 WHERE id = 5;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
UPDATE 1

COMMIT
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
BEGIN

 id | stock
----+-------
  5 |    22
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
COMMIT
```

Η Β είδε το απόθεμα αφού άλλαξε, 22 αντί για 25. Το lost update του κεφαλαίου 37 δεν μπορεί να συμβεί, γιατί η ανάγνωση και η εγγραφή έγιναν όσο κρατούσε η Α το lock.

Υπάρχουν τέσσερα είδη row lock, από το πιο αυστηρό στο πιο χαλαρό.

Lock	Ποιος το παίρνει	Εμποδίζει
`FOR UPDATE`	`DELETE`, `UPDATE` σε στήλη κλειδιού, `SELECT FOR UPDATE`	όλα τα άλλα row locks
`FOR NO KEY UPDATE`	`UPDATE` σε στήλες που δεν είναι κλειδιά	όλα εκτός από το `FOR KEY SHARE`
`FOR SHARE`	`SELECT FOR SHARE`	τα δύο πρώτα
`FOR KEY SHARE`	έλεγχος foreign key	μόνο το `FOR UPDATE`

Το `FOR KEY SHARE` εξηγεί γιατί μια νέα παραγγελία ενός πελάτη δεν περιμένει ένα ταυτόχρονο `UPDATE` στο email του ίδιου πελάτη. Ο έλεγχος του foreign key χρειάζεται μόνο να μη διαγραφεί ο πελάτης ή να μην αλλάξει το `id` του.

Να μην περιμένεις

Αν δεν θέλεις να περιμένεις, το `NOWAIT` αποτυγχάνει αμέσως όταν η γραμμή είναι κλειδωμένη. Η ρύθμιση `lock_timeout` ορίζει μέγιστο χρόνο αναμονής για κάθε lock.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
UPDATE products SET price = price WHERE id = 6;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
SELECT id FROM products WHERE id = 6 FOR UPDATE NOWAIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
ERROR:  could not obtain lock on row in relation "products"
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
ROLLBACK
```

Ουρές εργασιών με SKIP LOCKED

Μια πολύ συνηθισμένη χρήση των row locks είναι μια ουρά εργασιών μέσα στη βάση. Πολλοί workers παίρνουν εργασίες από έναν πίνακα, καθεμία πρέπει να εκτελεστεί μία φορά και κανένας worker δεν πρέπει να περιμένει τον άλλο. Το `SKIP LOCKED` προσπερνά τις γραμμές που έχει ήδη κλειδώσει κάποιος.

SQL

```
CREATE TABLE jobs (
  id         integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  kind       text NOT NULL,
  created_at timestamptz NOT NULL,
  done_at    timestamptz
);
INSERT INTO jobs (kind, created_at)
SELECT 'send_invoice', timestamptz '2026-06-30 10:00' + n * interval '1 minute'
FROM generate_series(1, 5) AS n;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 5
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
SELECT id, kind FROM jobs
WHERE done_at IS NULL
ORDER BY created_at
LIMIT 1
FOR UPDATE SKIP LOCKED;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

 id |     kind
----+--------------
  1 | send_invoice
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
BEGIN;
SELECT id, kind FROM jobs
WHERE done_at IS NULL
ORDER BY created_at
LIMIT 1
FOR UPDATE SKIP LOCKED;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
BEGIN

 id |     kind
----+--------------
  2 | send_invoice
(1 row)
```

Ο δεύτερος worker πήρε την εργασία 2 χωρίς να περιμένει, αφού η 1 ήταν κλειδωμένη. Κάθε worker εκτελεί την εργασία του, σημαδεύει τη γραμμή και κάνει commit.

SQL · ΣΥΝΕΔΡΙΑ Α

```
UPDATE jobs SET done_at = '2026-06-30 10:10' WHERE id = 1;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
UPDATE 1

COMMIT
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE jobs SET done_at = '2026-06-30 10:10' WHERE id = 2;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1

COMMIT
```

Αν ένας worker πέσει στη μέση, το transaction του αναιρείται, το lock ελευθερώνεται και η εργασία επιστρέφει στην ουρά. Αυτή η εγγύηση είναι ο λόγος που πολλές εφαρμογές κρατούν ουρές στην PostgreSQL αντί για ξεχωριστό σύστημα.

Deadlocks

Ένα **deadlock** συμβαίνει όταν δύο transactions περιμένουν το ένα το άλλο. Η Α κρατά το προϊόν 7 και θέλει το 8. Η Β κρατά το 8 και θέλει το 7. Καμία δεν μπορεί να προχωρήσει.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
UPDATE products SET stock = stock + 1 WHERE id = 7;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
BEGIN

UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
BEGIN;
UPDATE products SET stock = stock + 1 WHERE id = 8;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
BEGIN

UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
UPDATE products SET stock = stock + 1 WHERE id = 8;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
-- η εντολή περιμένει
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE products SET stock = stock + 1 WHERE id = 7;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
ERROR:  deadlock detected
DETAIL:  Process 43970 waits for ShareLock on transaction 3797; blocked by process 43971.
Process 43971 waits for ShareLock on transaction 3796; blocked by process 43970.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,8) in relation "products"
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
ROLLBACK
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
COMMIT
```

Η PostgreSQL δεν περιμένει για πάντα. Όταν ένα transaction περιμένει ένα lock περισσότερο από το `deadlock_timeout`, από προεπιλογή ένα δευτερόλεπτο, ελέγχει αν υπάρχει κύκλος αναμονής. Η Α περίμενε πρώτη, οπότε ο δικός της έλεγχος έτρεξε πρώτος, βρήκε τον κύκλο και ακύρωσε την ίδια. Το lock της γραμμής 7 ελευθερώθηκε και η Β ολοκληρώθηκε. Οι αριθμοί των διεργασιών και των transactions θα είναι άλλοι στη δική σου εκτέλεση. Το `DETAIL` του σφάλματος λέει ποιες διεργασίες και ποια transactions συμμετείχαν στον κύκλο. Το transaction της Α είναι πια σε κατάσταση σφάλματος και μπορεί μόνο να αναιρεθεί.

ΠΑΓΙΔΑ

Η συνηθισμένη αιτία deadlock είναι δύο διαδρομές κώδικα που κλειδώνουν τις ίδιες γραμμές με διαφορετική σειρά. Η λύση είναι να κλειδώνεις πάντα με την ίδια σειρά, για παράδειγμα κατά `id`. Ένα `SELECT ... WHERE id IN (...) ORDER BY id FOR UPDATE` στην αρχή του transaction κλειδώνει όλες τις γραμμές που θα χρειαστεί με σταθερή σειρά.

Table locks και αλλαγές σχήματος

Κάθε εντολή παίρνει και ένα lock σε επίπεδο πίνακα. Ένα `SELECT` παίρνει το πιο ελαφρύ, το `ACCESS SHARE`. Ένα `UPDATE` παίρνει `ROW EXCLUSIVE`. Τα δύο δεν συγκρούονται. Πολλές εντολές σχήματος όμως, όπως τα περισσότερα `ALTER TABLE`, θέλουν `ACCESS EXCLUSIVE`, που συγκρούεται με όλα, ακόμη και με το `SELECT`.

SQL · ΣΥΝΕΔΡΙΑ A

```
BEGIN;
SELECT count(*) FROM products;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ A

```
BEGIN

 count
-------
    30
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ B

```
SET lock_timeout = '500ms';
ALTER TABLE products ADD COLUMN weight_kg numeric;
RESET lock_timeout;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ B

```
SET

ERROR:  canceling statement due to lock timeout

RESET
```

SQL · ΣΥΝΕΔΡΙΑ A

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ A

```
COMMIT
```

Όσο ένα transaction που διάβασε τον πίνακα μένει ανοιχτό, το `ALTER TABLE` δεν μπορεί να πάρει το lock του. Χωρίς `lock_timeout` θα περίμενε. Χειρότερα, όσο περιμένει, κάθε νέο `SELECT` στον πίνακα μπαίνει στην ουρά πίσω του. Ένα ξεχασμένο transaction και ένα αθώο `ALTER TABLE` αρκούν για να σταματήσει η εφαρμογή. Γι’ αυτό οι αλλαγές σχήματος σε production τρέχουν πάντα με μικρό `lock_timeout` και ξαναδοκιμάζουν.

Ποιος περιμένει ποιον

Το `pg_stat_activity` δείχνει τι κάνει κάθε σύνδεση και το `pg_locks` ποια locks κρατά ή ζητά. Η συνάρτηση `pg_blocking_pids` επιστρέφει ποιες συνδέσεις εμποδίζουν μια άλλη.

SQL · ΣΥΝΕΔΡΙΑ A

```
BEGIN;
UPDATE products SET stock = stock WHERE id = 9;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ A

```
BEGIN

UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ B

```
UPDATE products SET stock = stock WHERE id = 9;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ B

```
-- η εντολή περιμένει
```

SQL · ΣΥΝΕΔΡΙΑ C

```
SELECT application_name, state, wait_event_type,
       cardinality(pg_blocking_pids(pid)) AS blocked_by
FROM pg_stat_activity
WHERE application_name LIKE 'book-%' AND datname = current_database()
ORDER BY application_name;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ C

```
 application_name |        state        | wait_event_type | blocked_by
------------------+---------------------+-----------------+------------
 book-a           | idle in transaction | Client          |          0
 book-b           | active              | Lock            |          1
 book-c           | active              | ⌀               |          0
(3 rows)
```

SQL · ΣΥΝΕΔΡΙΑ A

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ A

```
COMMIT
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1
```

Η συνεδρία Β περιμένει ένα `Lock` και την εμποδίζει μία άλλη σύνδεση, η Α, που είναι `idle in transaction`. Αυτή είναι η πρώτη ερώτηση όταν η εφαρμογή κολλάει. Ποιος κρατά lock και γιατί δεν τελειώνει το transaction του.

Advisory locks

Μερικές φορές θέλεις να κλειδώσεις κάτι που δεν είναι γραμμή, όπως «η διαδικασία τιμολόγησης του Ιουνίου». Τα **advisory locks** είναι locks πάνω σε αριθμούς που ορίζεις εσύ. Η βάση δεν ξέρει τι σημαίνουν. Εγγυάται μόνο ότι μία σύνδεση κάθε φορά κρατά ένα συγκεκριμένο κλειδί.

SQL · ΣΥΝΕΔΡΙΑ Α

```
SELECT pg_try_advisory_lock(202606) AS got_it;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
 got_it
--------
 t
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
SELECT pg_try_advisory_lock(202606) AS got_it;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
 got_it
--------
 f
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
SELECT pg_advisory_unlock(202606) AS released;
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Α

```
 released
----------
 t
(1 row)
```

Το `pg_try_advisory_lock` δεν περιμένει. Επιστρέφει αληθές αν πήρε το lock και ψευδές αν το κρατά άλλη σύνδεση. Το `pg_advisory_xact_lock` περιμένει και ελευθερώνεται αυτόματα στο τέλος του transaction, κάτι που το κάνει πιο ασφαλές.

ΚΡΑΤΑ ΑΥΤΟ

Τα `UPDATE` και `DELETE` κλειδώνουν γραμμές ως το τέλος του transaction. Το `FOR UPDATE` κλειδώνει στην ανάγνωση, το `SKIP LOCKED` φτιάχνει ουρές και το `NOWAIT` και το `lock_timeout` περιορίζουν την αναμονή. Τα deadlocks ανιχνεύονται και ακυρώνουν ένα transaction. Κλείδωνε με σταθερή σειρά. Οι αλλαγές σχήματος θέλουν `ACCESS EXCLUSIVE` και μικρό `lock_timeout`.

Ασκήσεις

ΑΣΚΗΣΗ 38.1 Διάβασε, αποφάσισε, γράψε

Σε transaction, κλείδωσε με `FOR UPDATE` το προϊόν 10 και διάβασε το απόθεμά του. Αν είναι τουλάχιστον 2, μείωσέ το κατά 2 με ένα `UPDATE` που επιστρέφει το νέο απόθεμα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | stock
----+-------
 10 |     6
(1 row)

 stock
-------
     4
(1 row)

UPDATE 1
```

ΑΣΚΗΣΗ 38.2 Τα locks ενός transaction

Σε transaction, ενημέρωσε το προϊόν 11 και εμφάνισε από το `pg_locks` τα locks που κρατά η σύνδεσή σου στον πίνακα `products`, με το είδος και το όνομα του πίνακα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
UPDATE 1

 locktype | relation |       mode       | granted
----------+----------+------------------+---------
 relation | products | RowExclusiveLock | t
(1 row)
```

ΑΣΚΗΣΗ 38.3 Worker ουράς

Με τον πίνακα `jobs` της θεωρίας, γράψε σε ένα query την κίνηση ενός worker. Πάρε τις δύο παλαιότερες εργασίες που δεν έχουν γίνει με `SKIP LOCKED`, σημάδεψέ τες με `done_at` ίσο με `2026-06-30 11:00` και επέστρεψε τα `id` τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id
----
  3
  4
(2 rows)

UPDATE 2
```

ΑΣΚΗΣΗ 38.4 Advisory lock ανά transaction

Σε transaction, πάρε το advisory lock με κλειδί 42 με `pg_advisory_xact_lock`. Εμφάνισε από το `pg_locks` το lock με `locktype` ίσο με `advisory` της σύνδεσής σου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pg_advisory_xact_lock
-----------------------

(1 row)

 locktype | objid |     mode      | granted
----------+-------+---------------+---------
 advisory |    42 | ExclusiveLock | t
(1 row)
```

ΑΣΚΗΣΗ 38.5 Όριο αναμονής

Σε transaction, όρισε `lock_timeout` 2 δευτερόλεπτα με `SET LOCAL` και εμφάνισε την τιμή του. Μετά το τέλος του transaction εμφάνισέ την ξανά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET

 lock_timeout
--------------
 2s
(1 row)

COMMIT

 lock_timeout
--------------
 0
(1 row)
```

ΚΕΦΑΛΑΙΟ 39

Functions, procedures και triggers

Η PostgreSQL εκτελεί και κώδικα, όχι μόνο queries. Μια function δίνει όνομα σε ένα query ή σε λογική με συνθήκες και βρόχους. Ένα trigger εκτελεί μια function αυτόματα όταν αλλάζει ένας πίνακας. Είναι ισχυρά εργαλεία και κρύβουν λογική, οπότε τα χρησιμοποιείς όπου η εγγύηση αξίζει την απόκρυψη.

SQL functions

Μια function σε γλώσσα `sql` είναι ένα query με παραμέτρους και όνομα. Η αξία μιας παραγγελίας, που γράψαμε σε πολλά κεφάλαια, γίνεται function.

SQL

```
CREATE FUNCTION order_total(p_order_id integer)
RETURNS numeric
LANGUAGE sql
STABLE
RETURN (SELECT COALESCE(sum(quantity * unit_price), 0)
        FROM order_items
        WHERE order_id = p_order_id);

SELECT id, status, order_total(id) AS total
FROM orders
WHERE customer_id = 13
ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 id |  status   |  total
----+-----------+---------
 20 | delivered |  114.90
 35 | delivered |  690.80
 40 | delivered | 2527.50
 41 | delivered |  289.00
 82 | delivered |  164.00
(5 rows)
```

Η μορφή με `RETURN` γράφει το σώμα ως κανονική SQL, που η βάση ελέγχει όταν φτιάχνεται η function και ξέρει από ποιους πίνακες εξαρτάται. Υπάρχει και η παλαιότερη μορφή με το σώμα σε κείμενο μέσα σε `$$ ... $$`, που θα δεις συχνά.

Το `STABLE` είναι η **volatility** της function, μια υπόσχεση προς τον planner.

Κατηγορία	Υπόσχεση	Παράδειγμα
`IMMUTABLE`	ίδιο αποτέλεσμα για ίδια ορίσματα, για πάντα	`lower`, αριθμητικές πράξεις
`STABLE`	ίδιο αποτέλεσμα μέσα στο ίδιο query	διαβάζει πίνακες, `now()`
`VOLATILE`	μπορεί να αλλάζει σε κάθε κλήση	`random()`, αλλαγές δεδομένων

Μόνο οι `IMMUTABLE` functions μπορούν να μπουν σε index πάνω σε έκφραση ή σε generated column. Μια ψεύτικη υπόσχεση οδηγεί σε λάθος αποτελέσματα, οπότε η ασφαλής προεπιλογή, αν δεν δηλώσεις τίποτα, είναι το `VOLATILE`.

Functions που επιστρέφουν πίνακα

Με `RETURNS TABLE` μια function επιστρέφει γραμμές και μπαίνει στο `FROM` όπως ένας πίνακας.

SQL

```
CREATE FUNCTION customer_orders(p_customer_id integer)
RETURNS TABLE (order_id integer, ordered_on date, total numeric)
LANGUAGE sql
STABLE
BEGIN ATOMIC
  SELECT o.id, o.created_at::date, order_total(o.id)
  FROM orders AS o
  WHERE o.customer_id = p_customer_id
  ORDER BY o.created_at;
END;

SELECT * FROM customer_orders(13);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 order_id | ordered_on |  total
----------+------------+---------
       20 | 2025-05-04 |  114.90
       35 | 2025-09-03 |  690.80
       40 | 2025-09-19 | 2527.50
       41 | 2025-09-21 |  289.00
       82 | 2026-02-04 |  164.00
(5 rows)
```

Το `BEGIN ATOMIC ... END` είναι η μορφή του σώματος όταν περιέχει ολόκληρες εντολές. Μια function με ένα απλό `SELECT` όπως αυτή, η PostgreSQL μπορεί να την ενσωματώσει στο query που την καλεί, ώστε ο planner να σπρώξει φίλτρα μέσα της.

PL/pgSQL

Όταν χρειάζεσαι μεταβλητές, συνθήκες, βρόχους ή σφάλματα, γράφεις σε **PL/pgSQL**, τη διαδικαστική γλώσσα της PostgreSQL. Το σώμα είναι κείμενο μέσα σε `$$`, με ενότητα `DECLARE` για μεταβλητές και `BEGIN ... END`.

```sql
CREATE FUNCTION change_category_prices(p_category_id integer, p_pct numeric)
RETURNS integer
LANGUAGE plpgsql
AS $$
DECLARE
  v_changed integer;
BEGIN
  IF p_pct < -50 OR p_pct > 50 THEN
    RAISE EXCEPTION 'Μη αποδεκτή αλλαγή τιμής: % %%', p_pct;
  END IF;

  UPDATE products
  SET price = round(price * (1 + p_pct / 100), 2)
  WHERE category_id = p_category_id AND active;

  GET DIAGNOSTICS v_changed = ROW_COUNT;
  RAISE NOTICE 'Άλλαξαν % προϊόντα', v_changed;
  RETURN v_changed;
END;
$$;

SELECT change_category_prices(12, 10);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

NOTICE:  Άλλαξαν 3 προϊόντα
 change_category_prices
------------------------
                      3
(1 row)
```

Το `RAISE NOTICE` στέλνει μήνυμα στον client, όπως φαίνεται πάνω από το αποτέλεσμα. Το `RAISE EXCEPTION` σταματά τη function και ακυρώνει το transaction, όπως κάθε σφάλμα. Το `%` στο κείμενο αντικαθίσταται από την επόμενη τιμή και το `%%` τυπώνει το ίδιο το σύμβολο.

```sql
SELECT change_category_prices(12, 80);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  Μη αποδεκτή αλλαγή τιμής: 80 %
CONTEXT:  PL/pgSQL function change_category_prices(integer,numeric) line 6 at RAISE
```

Μια function εκτελείται μέσα στο transaction του query που την καλεί. Αν αποτύχει, αναιρούνται όλα όσα έκανε μαζί με το υπόλοιπο transaction.

Procedures

Μια **procedure** μοιάζει με function αλλά καλείται με `CALL` και μπορεί να κάνει `COMMIT` μέσα της. Είναι χρήσιμη για μεγάλες εργασίες που χωρίζονται σε κομμάτια, ώστε κάθε κομμάτι να γίνεται commit χωριστά και να μην κρατιούνται locks για ώρα.

SQL

```
CREATE TABLE events_archive (LIKE events);

CREATE PROCEDURE archive_events(p_before timestamptz, p_batch integer)
LANGUAGE plpgsql
AS $$
DECLARE
  v_moved integer;
BEGIN
  LOOP
    WITH moved AS (
      DELETE FROM events
      WHERE id IN (SELECT id FROM events WHERE occurred_at < p_before LIMIT p_batch)
      RETURNING *
    )
    INSERT INTO events_archive SELECT * FROM moved;
    GET DIAGNOSTICS v_moved = ROW_COUNT;
    EXIT WHEN v_moved = 0;
    RAISE NOTICE 'Μεταφέρθηκαν % events', v_moved;
    COMMIT;
  END LOOP;
END;
$$;

CALL archive_events('2025-03-01', 20);
SELECT count(*) AS archived FROM events_archive;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

CREATE PROCEDURE

NOTICE:  Μεταφέρθηκαν 20 events
NOTICE:  Μεταφέρθηκαν 20 events
NOTICE:  Μεταφέρθηκαν 10 events
CALL

 archived
----------
       50
(1 row)
```

Κάθε κομμάτι των 20 γραμμών έγινε commit πριν από το επόμενο. Αν η διαδικασία διακοπεί στη μέση, όσα κομμάτια ολοκληρώθηκαν μένουν. Το `CALL` με `COMMIT` μέσα δεν μπορεί να εκτελεστεί μέσα σε ανοιχτό transaction.

Triggers

Ένα **trigger** συνδέει μια function με ένα γεγονός σε έναν πίνακα. Η function του trigger γράφεται σε PL/pgSQL, επιστρέφει `trigger` και έχει πρόσβαση στις μεταβλητές `NEW` και `OLD`, τη νέα και την παλιά έκδοση της γραμμής.

Ένα **BEFORE** trigger εκτελείται πριν γραφτεί η γραμμή και μπορεί να την αλλάξει. Η κλασική χρήση είναι μια στήλη που κρατά πότε άλλαξε τελευταία φορά η γραμμή.

SQL

```
ALTER TABLE products ADD COLUMN updated_at timestamptz;

CREATE FUNCTION touch_updated_at()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
  NEW.updated_at := statement_timestamp();
  RETURN NEW;
END;
$$;

CREATE TRIGGER products_touch
BEFORE UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION touch_updated_at();

UPDATE products SET stock = stock + 1 WHERE id = 1;
SELECT id, updated_at IS NOT NULL AS touched FROM products WHERE id IN (1, 2) ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

CREATE FUNCTION

CREATE TRIGGER

UPDATE 1

 id | touched
----+---------
  1 | t
  2 | f
(2 rows)
```

Η function επιστρέφει τη γραμμή που θα γραφτεί. Αν επέστρεφε `NULL`, η αλλαγή αυτής της γραμμής θα ακυρωνόταν σιωπηλά. Το `FOR EACH ROW` εκτελεί τη function μία φορά για κάθε γραμμή που αλλάζει.

Ένα **AFTER** trigger εκτελείται αφού γραφτεί η γραμμή. Δεν μπορεί να την αλλάξει, αλλά μπορεί να γράψει αλλού, για παράδειγμα σε ιστορικό αλλαγών. Με `WHEN` εκτελείται μόνο όταν ισχύει μια συνθήκη.

SQL

```
CREATE TABLE price_history (
  product_id integer NOT NULL,
  old_price  numeric(10,2) NOT NULL,
  new_price  numeric(10,2) NOT NULL,
  changed_by text NOT NULL DEFAULT current_user
);

CREATE FUNCTION log_price_change()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
```

```
  INSERT INTO price_history (product_id, old_price, new_price)
  VALUES (NEW.id, OLD.price, NEW.price);
  RETURN NULL;
END;
$$;

CREATE TRIGGER products_price_history
AFTER UPDATE OF price ON products
FOR EACH ROW
WHEN (OLD.price IS DISTINCT FROM NEW.price)
EXECUTE FUNCTION log_price_change();

UPDATE products SET price = price * 1.05 WHERE category_id = 13;
UPDATE products SET stock = stock + 1 WHERE category_id = 13;
SELECT product_id, old_price, new_price FROM price_history ORDER BY product_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

CREATE FUNCTION

CREATE TRIGGER

UPDATE 3

UPDATE 3

 product_id | old_price | new_price
------------+-----------+-----------
         25 |    289.00 |    303.45
         26 |    459.00 |    481.95
         27 |     42.00 |     44.10
(3 rows)
```

Το δεύτερο `UPDATE` δεν άλλαξε τιμές, οπότε το trigger δεν έγραψε τίποτα. Η επιστρεφόμενη τιμή ενός AFTER trigger αγνοείται και συνηθίζεται το `NULL`.

ΠΑΓΙΔΑ

Τα triggers κάνουν δουλειά που δεν φαίνεται στο query που τα ενεργοποιεί. Όποιος διαβάζει το `UPDATE` δεν ξέρει ότι γράφεται και ιστορικό. Ένα trigger που αλλάζει άλλους πίνακες, που μπορεί να έχουν δικά τους triggers, γίνεται γρήγορα δύσκολο να παρακολουθήσεις. Χρησιμοποίησέ τα για εγγυήσεις που πρέπει να ισχύουν όποιος κι αν γράφει στον πίνακα, όπως ιστορικό και συνέπεια δεδομένων. Την υπόλοιπη λογική κράτησέ τη στην εφαρμογή.

ΚΡΑΤΑ ΑΥΤΟ

Οι SQL functions δίνουν όνομα σε queries και δηλώνουν volatility. Η PL/pgSQL προσθέτει μεταβλητές, συνθήκες, βρόχους και `RAISE`. Οι procedures μπορούν να κάνουν `COMMIT` ανάμεσα σε κομμάτια δουλειάς. Τα BEFORE triggers αλλάζουν τη γραμμή πριν γραφτεί και τα AFTER γράφουν αλλού αφού γραφτεί.

Ασκήσεις

ΑΣΚΗΣΗ 39.1 Ονοματεπώνυμο ως function

Φτιάξε `IMMUTABLE` SQL function `full_name(first text, last text)` που επιστρέφει όνομα και επώνυμο με ένα κενό ανάμεσα. Χρησιμοποίησέ τη για τους πελάτες 1 έως 3.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 id |      full_name
----+---------------------
  1 | Μαρία Παπαδοπούλου
  2 | Γιώργος Παπαδόπουλος
  3 | Ελένη Νικολάου
(3 rows)
```

ΑΣΚΗΣΗ 39.2 Κορυφαία προϊόντα κατηγορίας

Φτιάξε function `top_products(p_category integer, p_n integer)` που επιστρέφει πίνακα με όνομα και έσοδα των `p_n` προϊόντων της κατηγορίας με τα περισσότερα έσοδα από παραγγελίες που δεν ακυρώθηκαν ούτε επιστράφηκαν. Κάλεσέ τη για την κατηγορία 9 και δύο προϊόντα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

         name          | revenue
-----------------------+---------
 Οθόνη 27" 4K          | 3948.00
 USB-C docking station | 1112.00
(2 rows)
```

ΑΣΚΗΣΗ 39.3 Email πάντα σε πεζά

Φτιάξε BEFORE trigger στον `customers` που μετατρέπει το email σε πεζά σε κάθε `INSERT` και `UPDATE`. Πρόσθεσε πελάτη με email `Nikos.Test@Example.GR` και επέστρεψε το email που αποθηκεύτηκε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

CREATE TRIGGER

         email
----------------------
 nikos.test@example.gr
(1 row)

INSERT 0 1
```

ΑΣΚΗΣΗ 39.4 Απόθεμα που δεν γίνεται αρνητικό

Φτιάξε AFTER INSERT trigger στον `order_items` που μειώνει το απόθεμα του προϊόντος κατά την ποσότητα. Αν το απόθεμα δεν φτάνει, να σταματά με `RAISE EXCEPTION`. Δοκίμασέ το με δύο τεμάχια του προϊόντος 26 στην παραγγελία 162 και μετά με δέκα τεμάχια του προϊόντος 25 στην ίδια παραγγελία.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

CREATE TRIGGER

INSERT 0 1

 stock
-------
     2
(1 row)

ERROR:  Δεν υπάρχει αρκετό απόθεμα για το προϊόν 25
CONTEXT:  PL/pgSQL function reserve_stock() line 6 at RAISE
```

ΑΣΚΗΣΗ 39.5 Μέτρηση με procedure

Φτιάξε procedure `deactivate_empty(INOUT p_count integer DEFAULT 0)` που απενεργοποιεί τα ενεργά προϊόντα με μηδενικό απόθεμα και επιστρέφει πόσα απενεργοποίησε. Κάλεσέ τη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE PROCEDURE

 p_count
---------
       2
(1 row)
```

ΚΕΦΑΛΑΙΟ 40

Range types, exclusion constraints και partitioning

Πολλά δεδομένα είναι διαστήματα. Μια προσφορά ισχύει από μια στιγμή ως μια άλλη, μια κράτηση πιάνει κάποιες ημέρες. Οι range types τα κρατούν ως μία τιμή και τα exclusion constraints εγγυώνται ότι δεν επικαλύπτονται. Το partitioning χωρίζει έναν μεγάλο πίνακα σε κομμάτια κατά διάστημα τιμών.

Range types

Ένα **range** είναι ένα διάστημα τιμών ενός τύπου. Η PostgreSQL έχει έτοιμους τους `int4range`, `numrange`, `daterange` και `tstzrange`. Γράφεται ως κείμενο με αγκύλη για κλειστό άκρο και παρένθεση για ανοιχτό. Το `[2026-06-01,2026-07-01)` είναι ο Ιούνιος ως μισάνοιχτο διάστημα, όπως στο κεφάλαιο 6.

SQL

```
SELECT daterange('2026-06-01', '2026-07-01') AS june,
       daterange('2026-06-01', '2026-07-01') @> date '2026-06-30' AS has_30th,
       daterange('2026-06-01', '2026-07-01') && daterange('2026-06-25', '2026-07-05') AS overlaps,
       daterange('2026-06-01', '2026-07-01') * daterange('2026-06-25', '2026-07-05') AS common,
       upper(daterange('2026-06-01', '2026-07-01')) AS ends;
```

ΑΠΟΤΕΛΕΣΜΑ

```
          june            | has_30th | overlaps |         common          |    ends
--------------------------+----------+----------+-------------------------+------------
 [2026-06-01,2026-07-01) | t        | t        | [2026-06-25,2026-07-01) | 2026-07-01
(1 row)
```

Ο constructor `daterange(αρχή, τέλος)` φτιάχνει από προεπιλογή μισάνοιχτο διάστημα. Ο `@>` ελέγχει αν το range περιέχει μια τιμή ή ένα άλλο range. Ο `&&` ελέγχει αν δύο ranges επικαλύπτονται. Ο `*` δίνει την τομή. Ένα άκρο που λείπει σημαίνει χωρίς όριο, όπως στο `[2026-06-01,)`.

Προσφορές σε διαστήματα

Το κατάστημα κάνει προσφορές σε προϊόντα για συγκεκριμένα διαστήματα. Ένα προϊόν δεν μπορεί να έχει δύο προσφορές ταυτόχρονα.

SQL

```
CREATE TABLE promotions (
  id          integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  product_id  integer NOT NULL REFERENCES products (id),
  discount    numeric(4,2) NOT NULL CHECK (discount > 0 AND discount < 1),
  during      tstzrange NOT NULL
);

INSERT INTO promotions (product_id, discount, during) VALUES
  (19, 0.20, tstzrange('2025-11-20', '2025-12-01')),
  (19, 0.10, tstzrange('2026-06-01', '2026-06-15')),
  (12, 0.15, tstzrange('2026-05-01', '2026-06-01'));
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 3
```

Ένα `UNIQUE` δεν μπορεί να εκφράσει τον κανόνα «καμία επικάλυψη για το ίδιο προϊόν». Δύο διαστήματα μπορεί να είναι διαφορετικά και να επικαλύπτονται. Εδώ χρειάζεται ένα **exclusion constraint**. Γενικεύει το `UNIQUE`. Αντί να απαγορεύει δύο γραμμές με ίσες τιμές, απαγορεύει δύο γραμμές για τις οποίες ένας τελεστής είναι αληθής.

SQL

```
CREATE EXTENSION IF NOT EXISTS btree_gist;

ALTER TABLE promotions
  ADD CONSTRAINT promotions_no_overlap
  EXCLUDE USING gist (product_id WITH =, during WITH &&);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

ALTER TABLE
```

Ο κανόνας διαβάζεται ως εξής. Δεν επιτρέπονται δύο γραμμές με ίσο `product_id` και διαστήματα που επικαλύπτονται. Το exclusion constraint στηρίζεται σε GiST index, όπως το `UNIQUE` σε B-tree. Η επέκταση `btree_gist` επιτρέπει στο GiST να συγκρίνει και απλές τιμές με `=`.

SQL

```
INSERT INTO promotions (product_id, discount, during)
VALUES (19, 0.25, tstzrange('2026-06-10', '2026-06-20'));
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  conflicting key value violates exclusion constraint "promotions_no_overlap"
DETAIL:  Key (product_id, during)=(19, ["2026-06-10 00:00:00+03","2026-06-20
00:00:00+03")) conflicts with existing key (product_id, during)=(19, ["2026-06-01
00:00:00+03","2026-06-15 00:00:00+03")).
```

Η βάση απέρριψε την προσφορά γιατί ξεκινά πριν τελειώσει η προηγούμενη. Μια προσφορά από τις 15 Ιουνίου θα γινόταν δεκτή, αφού τα μισάνοιχτα διαστήματα `[1,15)` και `[15,20)` δεν επικαλύπτονται. Ο ίδιος κανόνας χωρίς constraint θα χρειαζόταν έλεγχο στην εφαρμογή, που δεν αντέχει ταυτόχρονες εισαγωγές, όπως είδες στο κεφάλαιο 37.

> **ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL**
>
> Η PostgreSQL 18 έχει και **temporal constraints**. Ένα `PRIMARY KEY (product_id, during WITHOUT OVERLAPS)` εκφράζει τον ίδιο κανόνα με σύνταξη του προτύπου και μπορεί να αποτελέσει στόχο για temporal foreign keys με `PERIOD`.

Ποια προσφορά ίσχυε

Με τον `@>` βρίσκεις ποια προσφορά ίσχυε τη στιγμή μιας παραγγελίας.

SQL

```
SELECT o.id AS order_id, o.created_at::date, p.name, promo.discount
FROM orders AS o
JOIN order_items AS oi ON oi.order_id = o.id
JOIN products AS p ON p.id = oi.product_id
JOIN promotions AS promo ON promo.product_id = oi.product_id
                        AND promo.during @> o.created_at
ORDER BY o.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | created_at |     name      | discount
----------+------------+---------------+----------
       58 | 2025-11-21 | Ακουστικά ANC |     0.20
      114 | 2026-05-04 | Laptop Pro 16 |     0.15
      130 | 2026-05-26 | Laptop Pro 16 |     0.15
      143 | 2026-06-13 | Ακουστικά ANC |     0.10
(4 rows)
```

Το join με συνθήκη περιεχομένου είναι το join με εύρος του κεφαλαίου 9, με την ευκολία ότι το διάστημα είναι μία τιμή. Το GiST index του exclusion constraint βοηθά και σε αυτό το query.

Partitioning

Όταν ένας πίνακας γίνεται πολύ μεγάλος, μπορείς να τον χωρίσεις σε **partitions**. Ο γονικός πίνακας δεν κρατά δεδομένα. Κάθε partition είναι ένας κανονικός πίνακας με ένα διάστημα τιμών του **partition key**. Τα queries γράφονται στον γονικό και η βάση στέλνει κάθε γραμμή στο σωστό partition.

SQL

```
CREATE TABLE events_log (
  id          bigint NOT NULL,
  customer_id integer NOT NULL,
  kind        text NOT NULL,
  occurred_at timestamptz NOT NULL
) PARTITION BY RANGE (occurred_at);

CREATE TABLE events_log_2025h1 PARTITION OF events_log
  FOR VALUES FROM ('2025-01-01') TO ('2025-07-01');
CREATE TABLE events_log_2025h2 PARTITION OF events_log
  FOR VALUES FROM ('2025-07-01') TO ('2026-01-01');
CREATE TABLE events_log_2026h1 PARTITION OF events_log
  FOR VALUES FROM ('2026-01-01') TO ('2026-07-01');

INSERT INTO events_log SELECT id, customer_id, kind, occurred_at FROM events;

SELECT tableoid::regclass AS partition, count(*)
FROM events_log
GROUP BY 1
ORDER BY 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

CREATE TABLE

CREATE TABLE

CREATE TABLE

INSERT 0 1395

     partition     | count
-------------------+-------
 events_log_2025h1 |   210
 events_log_2025h2 |   431
 events_log_2026h1 |   754
(3 rows)
```

Η ψευδοστήλη `tableoid` λέει σε ποιον πίνακα βρίσκεται κάθε γραμμή. Τα όρια `FROM ... TO` είναι μισάνοιχτα, οπότε τα partitions κολλάνε χωρίς κενά και χωρίς επικαλύψεις.

Το πρώτο όφελος είναι το **partition pruning**. Όταν η συνθήκη ενός query αφορά το partition key, η βάση διαβάζει μόνο τα partitions που μπορεί να περιέχουν γραμμές.

SQL

```
EXPLAIN (COSTS OFF)
SELECT count(*) FROM events_log
WHERE occurred_at >= '2026-03-01' AND occurred_at < '2026-04-01';
```

ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate
  ->  Seq Scan on events_log_2026h1 events_log
        Filter: ((occurred_at >= '2026-03-01 00:00:00+02'::timestamp with time zone) AND
(occurred_at < '2026-04-01 00:00:00+03'::timestamp with time zone))
```

Το plan αγγίζει μόνο το `events_log_2026h1`. Το δεύτερο όφελος είναι η διαγραφή παλιών δεδομένων. Αντί για ένα τεράστιο `DELETE` που αφήνει εκατομμύρια dead tuples, αποσπάς το partition με `DETACH PARTITION` και το σβήνεις ή το αρχειοθετείς.

SQL

```
ALTER TABLE events_log DETACH PARTITION events_log_2025h1;

SELECT min(occurred_at)::date AS first_day, count(*) FROM events_log;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

 first_day  | count
------------+-------
 2025-07-05 |  1185
(1 row)
```

Μια γραμμή με τιμή έξω από όλα τα partitions απορρίπτεται. Ένα `DEFAULT` partition μαζεύει όσες δεν ταιριάζουν αλλού.

SQL

```
INSERT INTO events_log VALUES (999999, 1, 'view', '2026-07-15');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  no partition of relation "events_log" found for row
DETAIL:  Partition key of the failing row contains (occurred_at) = (2026-07-15 00:00:00+03).
```

ΠΑΓΙΔΑ

Το partitioning δεν κάνει αυτόματα τα queries γρηγορότερα. Ένα query που δεν φιλτράρει με το partition key διαβάζει όλα τα partitions και πληρώνει επιπλέον κόστος σχεδιασμού. Ένα primary key ή unique constraint πρέπει να περιέχει το partition key. Κάνε partitioning σε πίνακες εκατοντάδων εκατομμυρίων γραμμών με σαφή στήλη χρόνου και ανάγκη να σβήνονται παλιά δεδομένα. Για μικρότερους πίνακες ένα καλό index είναι σχεδόν πάντα αρκετό.

ΚΡΑΤΑ ΑΥΤΟ

Τα ranges κρατούν διαστήματα ως μία τιμή, με `@>` για περιεχόμενο και `&&` για επικάλυψη. Ένα exclusion constraint με GiST απαγορεύει επικαλύψεις. Το partitioning χωρίζει έναν πίνακα κατά διαστήματα του partition key, επιτρέπει pruning στα queries και διαγραφή παλιών δεδομένων με `DETACH`.

Ασκήσεις

ΑΣΚΗΣΗ 40.1 Πράξεις με διαστήματα

Για τα διαστήματα Μαΐου και Ιουνίου 2026 ως `daterange`, εμφάνισε αν επικαλύπτονται, αν είναι συνεχόμενα με τον τελεστή `-|-`, την ένωσή τους με `+` και τον αριθμό των ημερών της ένωσης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 overlap | adjacent |        together         | days
---------+----------+-------------------------+------
 f       | t        | [2026-05-01,2026-07-01) |   61
(1 row)
```

ΑΣΚΗΣΗ 40.2 Επιτρεπτή προσφορά

Πρόσθεσε μια προσφορά 25% στο προϊόν 19 από τις 15 Ιουνίου 2026 ως την 1η Ιουλίου 2026. Μετά εμφάνισε όλες τις προσφορές του προϊόντος 19 κατά ημερομηνία έναρξης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1

 discount |                       during
----------+-----------------------------------------------------
     0.20 | ["2025-11-20 00:00:00+02","2025-12-01 00:00:00+02")
     0.10 | ["2026-06-01 00:00:00+03","2026-06-15 00:00:00+03")
     0.25 | ["2026-06-15 00:00:00+03","2026-07-01 00:00:00+03")
(3 rows)
```

ΑΣΚΗΣΗ 40.3 Τιμή μετά την προσφορά

Για κάθε γραμμή παραγγελίας που αγοράστηκε ενώ ίσχυε προσφορά, εμφάνισε `order_id`, `product_id`, την τιμή μονάδας και την τιμή που θα είχε μετά την έκπτωση, στρογγυλεμένη σε δύο δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 order_id | product_id | unit_price | discounted
----------+------------+------------+------------
       58 |         19 |     199.00 |     159.20
      114 |         12 |    2199.00 |    1869.15
      130 |         12 |    2199.00 |    1869.15
      143 |         19 |     199.00 |     179.10
(4 rows)
```

ΑΣΚΗΣΗ 40.4 Pruning σε δύο partitions

Δείξε με `EXPLAIN (COSTS OFF)` το plan για τα events του `events_log` από τις 15 Δεκεμβρίου 2025 ως τις 15 Ιανουαρίου 2026.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate
  ->  Append
        ->  Seq Scan on events_log_2025h2 events_log_1
              Filter: ((occurred_at >= '2025-12-15 00:00:00+02'::timestamp with time
zone) AND (occurred_at < '2026-01-15 00:00:00+02'::timestamp with time zone))
        ->  Seq Scan on events_log_2026h1 events_log_2
              Filter: ((occurred_at >= '2025-12-15 00:00:00+02'::timestamp with time
zone) AND (occurred_at < '2026-01-15 00:00:00+02'::timestamp with time zone))
```

ΑΣΚΗΣΗ 40.5 Partition για τα υπόλοιπα

Πρόσθεσε `DEFAULT` partition στον `events_log` και ξαναδοκίμασε την εισαγωγή του event της 15ης Ιουλίου 2026. Εμφάνισε σε ποιο partition κατέληξε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

     partition
____________________
 events_log_default
(1 row)

INSERT 0 1
```

ΚΕΦΑΛΑΙΟ 41

Ρόλοι, δικαιώματα και Row Level Security

Κάθε σύνδεση στη βάση γίνεται ως κάποιος ρόλος και κάθε ρόλος μπορεί να κάνει μόνο όσα του έχουν επιτραπεί. Με τα δικαιώματα ορίζεις ποιους πίνακες και ποιες στήλες βλέπει ένας ρόλος. Με το Row Level Security ορίζεις και ποιες γραμμές.

Ρόλοι

Στην PostgreSQL χρήστες και ομάδες είναι το ίδιο πράγμα, **ρόλοι**. Ένας ρόλος με `LOGIN` μπορεί να συνδεθεί. Ένας ρόλος χωρίς `LOGIN` είναι ομάδα δικαιωμάτων που δίνεται σε άλλους ρόλους. Οι ρόλοι ανήκουν σε όλο τον server, όχι σε μια βάση.

Μέχρι τώρα συνδεόσουν πιθανότατα ως **superuser**, έναν ρόλο που παρακάμπτει όλους τους ελέγχους. Μια εφαρμογή δεν πρέπει ποτέ να συνδέεται έτσι. Ας φτιάξουμε έναν ρόλο για αναφορές και έναν για την εφαρμογή.

SQL

```
CREATE ROLE book_reporting NOLOGIN;
CREATE ROLE book_analyst LOGIN PASSWORD 'change-me' IN ROLE book_reporting;
CREATE ROLE book_app LOGIN PASSWORD 'change-me';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE ROLE
CREATE ROLE
CREATE ROLE
```

Ο `book_analyst` είναι μέλος της ομάδας `book_reporting` και κληρονομεί ό,τι της επιτραπεί. Τα δικαιώματα τα δίνεις στην ομάδα, όχι σε κάθε άτομο. Όταν έρθει ένας νέος αναλυτής, αρκεί να γίνει μέλος.

GRANT και REVOKE

Ένας νέος ρόλος δεν μπορεί να διαβάσει κανέναν πίνακα. Το `GRANT` δίνει δικαιώματα και το `REVOKE` τα αφαιρεί. Τα βασικά δικαιώματα σε πίνακα είναι τα `SELECT`, `INSERT`, `UPDATE` και `DELETE`.

SQL

```
GRANT SELECT ON orders, order_items, products, categories TO book_reporting;
```

ΑΠΟΤΕΛΕΣΜΑ

```
GRANT
```

Για να δοκιμάσεις τι βλέπει ένας ρόλος, ένας superuser μπορεί να τον υποδυθεί με `SET ROLE`. Από εκεί και πέρα οι έλεγχοι γίνονται σαν να είχε συνδεθεί ο ίδιος ο ρόλος.

SQL

```
SET ROLE book_analyst;
SELECT count(*) FROM orders;
SELECT count(*) FROM customers;
RESET ROLE;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET

 count
-------
   162
(1 row)

ERROR:  permission denied for table customers

RESET
```

Ο αναλυτής διάβασε τις παραγγελίες, αλλά όχι τους πελάτες. Δεν είχε δοθεί δικαίωμα στον `customers`, που περιέχει προσωπικά δεδομένα.

Τα δικαιώματα δίνονται και ανά στήλη. Ο αναλυτής χρειάζεται την πόλη κάθε πελάτη για αναφορές ανά περιοχή, αλλά όχι το email ή το τηλέφωνο.

SQL

```
GRANT SELECT (id, city_id, created_at) ON customers TO book_reporting;

SET ROLE book_analyst;
SELECT city_id, count(*) FROM customers GROUP BY city_id ORDER BY city_id LIMIT 3;
SELECT email FROM customers LIMIT 1;
RESET ROLE;
```

ΑΠΟΤΕΛΕΣΜΑ

```
GRANT

SET

 city_id | count
---------+-------
       1 |     7
       2 |     6
       3 |     2
(3 rows)

ERROR:  permission denied for table customers

RESET
```

Δικαιώματα της εφαρμογής

Ο ρόλος της εφαρμογής χρειάζεται να διαβάζει και να γράφει, αλλά όχι να αλλάζει το σχήμα ή να διαγράφει πίνακες. Αυτά τα κάνουν μόνο οι migrations με έναν ξεχωριστό ρόλο.

SQL

```
GRANT SELECT, INSERT, UPDATE ON orders, order_items, payments TO book_app;
GRANT SELECT ON products, categories, customers, cities TO book_app;
GRANT USAGE ON SEQUENCE orders_id_seq, payments_id_seq TO book_app;
```

ΑΠΟΤΕΛΕΣΜΑ

```
GRANT

GRANT

GRANT
```

Το `USAGE` στις ακολουθίες χρειάζεται για να δίνουν αριθμούς στα identity columns κατά το `INSERT`. Ο `book_app` δεν έχει `DELETE` στον `orders`. Αν ένα λάθος στον κώδικα προσπαθήσει να διαγράψει παραγγελίες, η βάση θα αρνηθεί.

Ο κατάλογος `information_schema.role_table_grants` δείχνει ποια δικαιώματα έχει κάθε ρόλος.

SQL

```
SELECT table_name, string_agg(privilege_type, ', ' ORDER BY privilege_type) AS privileges
FROM information_schema.role_table_grants
WHERE grantee = 'book_app'
GROUP BY table_name
ORDER BY table_name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 table_name  |       privileges
-------------+------------------------
 categories  | SELECT
 cities      | SELECT
 customers   | SELECT
 order_items | INSERT, SELECT, UPDATE
 orders      | INSERT, SELECT, UPDATE
 payments    | INSERT, SELECT, UPDATE
 products    | SELECT
(7 rows)
```

ΠΑΓΙΔΑ

Τα δικαιώματα ισχύουν για όσους πίνακες υπάρχουν τη στιγμή του `GRANT`. Ένας νέος πίνακας δεν έχει δικαιώματα για κανέναν εκτός από τον ιδιοκτήτη του. Με `ALTER DEFAULT PRIVILEGES` ορίζεις δικαιώματα για πίνακες που θα φτιαχτούν στο μέλλον.

Row Level Security

Τα δικαιώματα λένε ποιους πίνακες βλέπει ένας ρόλος. Σε πολλές εφαρμογές όμως το ερώτημα είναι ποιες γραμμές. Ένας πελάτης πρέπει να βλέπει μόνο τις δικές του παραγγελίες. Το **Row Level Security**, RLS, προσθέτει σε κάθε query αυτόματα μια συνθήκη που ορίζει ποιες γραμμές είναι ορατές.

Μια συνηθισμένη πρακτική είναι η εφαρμογή να ορίζει στην αρχή κάθε transaction μια ρύθμιση με το `id` του χρήστη. Το RLS τη διαβάζει με `current_setting`.

SQL

```
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;

CREATE POLICY orders_own ON orders
  FOR ALL
  TO book_app
  USING (customer_id = NULLIF(current_setting('app.customer_id', true), '')::integer)
  WITH CHECK (customer_id = NULLIF(current_setting('app.customer_id', true), '')::integer);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

CREATE POLICY
```

Το `USING` ορίζει ποιες γραμμές είναι ορατές, στα `SELECT`, `UPDATE` και `DELETE`. Το `WITH CHECK` ορίζει ποιες γραμμές επιτρέπεται να γραφτούν, στα `INSERT` και `UPDATE`. Το δεύτερο όρισμα `true` του `current_setting` σημαίνει ότι αν η ρύθμιση δεν υπάρχει επιστρέφει `NULL` αντί για σφάλμα.

Το `NULLIF` καλύπτει μια λεπτομέρεια που θα σε έβρισκε στην πράξη. Αφού μια σύνδεση ορίσει μία φορά τη ρύθμιση με `SET LOCAL`, στο τέλος του transaction η ρύθμιση δεν εξαφανίζεται. Παίρνει την τιμή κενό κείμενο. Το `''::integer` είναι σφάλμα, οπότε το `NULLIF` μετατρέπει το κενό σε `NULL`.

```sql
BEGIN;
SET LOCAL ROLE book_app;
SET LOCAL app.customer_id = '13';
SELECT id, customer_id, status FROM orders ORDER BY id;
ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET

SET

 id | customer_id |  status
----+-------------+-----------
 20 |          13 | delivered
 35 |          13 | delivered
 40 |          13 | delivered
 41 |          13 | delivered
 82 |          13 | delivered
(5 rows)

ROLLBACK
```

Το query δεν έχει `WHERE`, αλλά ο `book_app` είδε μόνο τις παραγγελίες του πελάτη 13. Χωρίς ρύθμιση, η συνθήκη είναι `customer_id = NULL` και δεν βλέπει τίποτα. Αυτή είναι η ασφαλής προεπιλογή. Αν η εφαρμογή ξεχάσει να ορίσει τον πελάτη, δεν δείχνει τα δεδομένα όλων.

```sql
BEGIN;
SET LOCAL ROLE book_app;
SET LOCAL app.customer_id = '13';
INSERT INTO orders (customer_id, status, created_at) VALUES (14, 'pending', '2026-06-30');
ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET

SET

ERROR:  new row violates row-level security policy for table "orders"

ROLLBACK
```

Ο πελάτης 13 δεν μπορεί να φτιάξει παραγγελία για λογαριασμό του 14. Το `WITH CHECK` την απέρριψε.

> **ΕΙΔΙΚΑ ΣΤΗΝ POSTGRESQL**
> Ο ιδιοκτήτης ενός πίνακα και οι superusers παρακάμπτουν το RLS. Με `ALTER TABLE ... FORCE ROW LEVEL SECURITY` ισχύει και για τον ιδιοκτήτη. Ένας ρόλος με `BYPASSRLS` το παρακάμπτει πάντα, κάτι χρήσιμο για backups και εργασίες συντήρησης.

Views και functions με δικαιώματα

Ένα view εκτελείται από προεπιλογή με τα δικαιώματα του ιδιοκτήτη του. Έτσι ένα view μπορεί να δείχνει κομμάτι ενός πίνακα σε έναν ρόλο που δεν έχει δικαίωμα στον ίδιο τον πίνακα. Με `security_invoker = true` το view εκτελείται με τα δικαιώματα αυτού που το διαβάζει, ώστε να ισχύουν και οι policies του RLS για αυτόν.

Μια function με `SECURITY DEFINER` εκτελείται με τα δικαιώματα του ιδιοκτήτη της. Είναι ο τρόπος να επιτρέψεις μια συγκεκριμένη ενέργεια χωρίς να δώσεις γενικό δικαίωμα, για παράδειγμα μια function που ακυρώνει μια παραγγελία χωρίς ο ρόλος να έχει `UPDATE` στον πίνακα. Τέτοιες functions πρέπει να ορίζουν `SET search_path` και να ελέγχουν τα ορίσματά τους. Εκτελούνται με περισσότερα δικαιώματα από αυτόν που τις καλεί.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Οι ρόλοι είναι χρήστες και ομάδες. Δίνεις δικαιώματα σε ομάδες με `GRANT`, σε πίνακες, στήλες και ακολουθίες. Η εφαρμογή συνδέεται με ρόλο που κάνει μόνο όσα χρειάζεται. Το RLS προσθέτει συνθήκες ορατότητας ανά γραμμή με policies, με `USING` για ανάγνωση και `WITH CHECK` για εγγραφή.

Ασκήσεις

ΑΣΚΗΣΗ 41.1 Ρόλος μόνο για ανάγνωση

Φτιάξε ρόλο `book_viewer` χωρίς login με δικαίωμα ανάγνωσης στον `cities`. Υποδύσου τον με `SET ROLE` και δοκίμασε ένα `SELECT` και ένα `UPDATE`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE ROLE

GRANT

SET

 count
-------
    11
(1 row)

ERROR:  permission denied for table cities
```

ΑΣΚΗΣΗ 41.2 Ποιος έχει τι

Εμφάνισε από το `information_schema.role_table_grants` όλα τα δικαιώματα του `book_reporting` σε πίνακες, ένα ανά γραμμή, ταξινομημένα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 table_name  | privilege_type
-------------+----------------
 categories  | SELECT
 order_items | SELECT
 orders      | SELECT
 products    | SELECT
(4 rows)
```

ΑΣΚΗΣΗ 41.3 Κριτικές μόνο του εαυτού σου

Ενεργοποίησε RLS στον `reviews` με policy για τον `book_app` που επιτρέπει να βλέπει όλες τις κριτικές αλλά να αλλάζει μόνο όσες έγραψε ο πελάτης της ρύθμισης `app.customer_id`. Δώσε στον `book_app` `SELECT` και `UPDATE` στον πίνακα. Ως πελάτης 1, άλλαξε τον τίτλο όλων των κριτικών για το προϊόν 19 και δες πόσες άλλαξαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE
GRANT
CREATE POLICY
CREATE POLICY
SET
SET
 visible
---------
       4
(1 row)
UPDATE 1
```

ΑΣΚΗΣΗ 41.4 Χωρίς πελάτη, τίποτα

Ως `book_app` και χωρίς να ορίσεις `app.customer_id`, μέτρησε τις παραγγελίες που βλέπεις. Μετά όρισε τον πελάτη 6 και μέτρησε ξανά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SET
 without_customer
------------------
                 0
(1 row)
SET
 as_customer_6
---------------
            10
(1 row)
```

ΑΣΚΗΣΗ 41.5 View με τα δικαιώματα του αναγνώστη

Φτιάξε view `my_orders` πάνω στον `orders` με `security_invoker = true` και στήλες `id`, `status` και `created_at`. Δώσε `SELECT` στον `book_app`. Ως `book_app` και πελάτης 13, μέτρησε τις γραμμές του view.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE VIEW

GRANT

SET

SET

 count
-------
     5
(1 row)
```